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

Marleson Graf

Shared Memory Verification for Multicore Chip Designs

Florianópolis
2020

Marleson Graf

Shared Memory Verification for Multicore Chip Designs

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de mestre em Ciência da Computação.
Orientador: Prof. Luiz Cláudio Villar dos Santos, Dr.

Florianópolis
2020

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

Graf, Marleson
   Shared Memory Verification for Multicore Chip Designs / Marleson Graf ; orientador, Luiz Cláudio Villar dos Santos, 2020.
   81 p.

   Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2020.

   Inclui referências.

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

Marleson Graf

Shared Memory Verification for Multicore Chip Designs

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestre em Ciência da Computação.

Documento assinado digitalmente
Vania Bogorny
Data: 18/11/2020 08:35:25-0300
CPF: 684.586.570-15

Prof.ª Vania Bogorny, Dr.ª
Coordenadora do Programa

Documento assinado digitalmente
Luiz Claudio Villar dos Santos
Data: 17/11/2020 23:19:04-0300
CPF: 489.140.889-87

Prof. Luiz Cláudio Villar dos Santos, Dr.
Orientador

Florianópolis, 2020.

Ao Todo.

ACKNOWLEDGEMENTS

I would like to thank all colleagues, professors, and fellow researchers that contributed, direct or indirectly, to the development of this work, in special Olav P. Henschel and my advisor, Luiz C. V. dos Santos, whose prior work served as basis for the main contributions of this dissertation. I also would like to thank my family and friends for all support provided and encouragement over the years.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

“Any sufficiently advanced technology is indistinguishable from magic.”
(CLARKE, 1973)

RESUMO

Multiprocessadores em um único *chip*, conhecidos como *multicores*, normalmente proveem uma abstração de memória compartilhada coerente, a qual pode ser obtida por meio da implementação de protocolos de coerência de cache no *hardware*. Já foi mostrado que a coerência implementada em *hardware* pode ser escalável em relação ao aumento do número de núcleos. Além disso, arquiteturas *multicore* podem adotar grande relaxação das restrições de consistência sobre a atomicidade de escrita e sobre o ordenamento entre leituras e escritas. Como resultado, a validação de memória compartilhada em *multicore chips* enfrenta dois desafios principais: o maior número de comportamentos válidos de execução, resultante da relaxação da consistência, e o maior espaço de estados do protocolo de coerência, induzido pelo crescente número de núcleos. No nível microarquitetural, o esforço de validação baseia-se em verificação funcional contando com a simulação da representação de projeto. A geração de testes *litmus* é uma abordagem efetiva para expor erros de consistência, sem a necessidade de verificadores especializados, mas tem cobertura limitada quando usada para verificação funcional. A geração aleatória de testes é uma abordagem alternativa capaz de obter maior cobertura, mas requer um verificador independente para validar se os comportamentos de memória observados obedecem a um dado modelo de consistência de memória. Esta dissertação apresenta um *framework* de verificação de memória compartilhada e propõe uma nova abordagem como contribuição para um de seus módulos. O *framework* é resultado de um esforço de pesquisa coletivo, e utiliza geração aleatória de testes dirigida por cobertura. A abordagem proposta permite a personalização de verificadores eficientes de acordo com a arquitetura e microarquitetura alvos. Ela utiliza uma especificação abstrata do comportamento da memória compartilhada e um *template* de observabilidade para guiar a inserção de monitores em pontos apropriados da representação de projeto. Para garantir escalabilidade sem comprometer garantias de verificação, são definidos múltiplos monitores por domínio de núcleo. Ao contrário de verificadores convencionais, os quais são incapazes de lidar com o comportamento resultante da relaxação da atomicidade de escrita, a abordagem proposta permite a construção de verificadores compatíveis com ambos os tipos de atomicidade (relaxada ou estrita). Um verificador construído sob a nova abordagem foi comparado com um convencional, ambos dentro do mesmo *framework* de verificação. Eles foram avaliados através da execução dos mesmos conjuntos de testes, cada um composto por vários programas de tamanho fixo. Para programas de 4Ki instruções, o verificador convencional acusou falsos positivos para um terço dos conjuntos de testes, ao verificar projetos sem erros com 32 núcleos sob atomicidade relaxada de escrita, enquanto o novo verificador não acusou erro algum. Para projetos contendo erros, o novo verificador mostrou esforço adicional desprezível em alguns casos e frequentemente reduziu o esforço. Portanto, as evidências experimentais indicam que, sob a nova abordagem, o verificador é efetivo, frequentemente reduz o esforço para detectar erros e é adequado para verificar projetos com diferentes graus de relaxação da atomicidade de escrita.

Palavras-chave: *Multicore*. Memória compartilhada. Verificação. Geração aleatória de testes. Consistência. Coerência. Atomicidade de escrita.

RESUMO EXPANDIDO

Introdução

Multiprocessadores em um único *chip*, conhecidos como *multicores*, geralmente se baseiam na abstração de memória compartilhada, a qual costuma ser amparada por protocolos de coerência de cache. Há indicações de que, com decisões de projeto adequadas, o *hardware* de suporte à coerência pode ser mantido escalável com o aumento do número de núcleos. Como o número de núcleos ativos é termicamente limitado, espera-se que a coerência de cache continue desempenhando um papel dominante em *multicore chips* voltados a aplicações de propósitos gerais. Além disso, tem-se observado uma tendência em se relaxar as restrições de ordenamento entre leituras e escritas em memória compartilhada, que eram impostas pelo modelo sequencial de consistência com o objetivo de manter uma abstração simples para programação paralela, algo que não é mais mandatório, uma vez que a maioria dos programas utiliza sincronização. Essas tendências acabam aumentando a complexidade da validação do subsistema de memória compartilhada em *multicore chips* por duas razões principais: o maior número de comportamentos válidos (resultante da relaxação das regras de consistência) e o número crescente de estados do protocolo de coerência (induzido pelo aumento do número de núcleos). Embora a geração de testes *litmus* seja uma abordagem efetiva para a validação de um *multicore chip*, sua limitada cobertura restringe a verificação funcional (em tempo de projeto), a qual se baseia em simulação e não na direta execução de programas em protótipo. Por isso, a geração de testes aleatórios é a abordagem alternativa predominante para verificação funcional, pois obtém maior cobertura, mas requer uma ferramenta independente para verificar se os comportamentos observados obedecem a um dado modelo de consistência de memória. Essa verificação baseia-se na análise de *traces* de operações de memória, obtidos através de monitores em cada núcleo. Quando a observabilidade é limitada a um monitor por núcleo (tal como ocorre na validação de protótipo), essa análise resulta em um problema intratável. Por isso, verificadores a serem usados em tempo de projeto deveriam tirar vantagem da maior observabilidade da representação de um *multicore chip* e usar múltiplos monitores por núcleo, permitindo assim a escalabilidade do esforço sem comprometer as garantias de verificação. Além disso, verificadores convencionais assumem que as escritas sejam vistas como se fossem atômicas no nível arquitetural, embora elas sejam inerentemente não-atômicas quando observadas no nível microarquitetural, o que dificulta diferenciar comportamentos válidos de inválidos. Embora algumas arquiteturas tenham sido simplificadas para que o programador não se exponha a comportamentos não-atômicos (tais como ARMv8 e RISC-V), implementações agressivas dessas arquiteturas tendem a exibir comportamentos não-atômicos. Diante dessas implementações, verificadores que exploram maior observabilidade acabam sendo expostos a esses comportamentos, resultando em falsos positivos. Portanto, um novo verificador deveria ser capaz de explorar a maior observabilidade, sem incorrer em falsos positivos induzidos por comportamentos não-atômicos.

Objetivos

Esta dissertação apresenta parte de um esforço coletivo de pesquisa que visa dois objetivos finais: (1) o desenvolvimento de novos geradores de testes e (2) o desenvolvimento de novos verificadores de modelos de consistência de memória, ambos integrados em um mesmo *framework* de verificação. Visto que geradores de testes são necessários para avaliar verificadores, contribuições técnicas para atingir o primeiro objetivo são brevemente reportadas, as quais são parte do *framework* desenvolvido. Além disso, também é detalhada uma contribuição científica para atingir o segundo objetivo, que é o principal foco da dissertação. Essa contribuição

é uma continuação de um trabalho preliminar, o qual foi desenvolvido em conjunto por Olav P. Henschel e Luiz C. V. dos Santos em 2013, mas permaneceu não publicado. Esse trabalho consistia de um protótipo para um verificador atrelado a uma microarquitetura com comportamentos não-atômicos. Sua implementação estava incompleta e merecia validação experimental. Esse trabalho preliminar foi revisado, estendido, validado e publicado em co-autoria. O envolvimento do autor nessa contribuição inclui (1) a remodelagem dos algoritmos para a verificação dos axiomas propostos, (2) a extensão do *template* de observabilidade para capturar apropriadamente todos os eventos físicos necessários, e (3) a mudança de foco para combinar com as tendências arquiteturais atuais que simplificam a atomicidade de escrita.

Metodologia

A abordagem proposta permite a personalização de verificadores eficientes de acordo com a arquitetura e microarquitetura alvos. Ela utiliza uma especificação abstrata do comportamento da memória compartilhada e um *template* de observabilidade para guiar a inserção de monitores em pontos apropriados da representação de projeto. Para a avaliação e validação da abordagem proposta, foram adotados os seguintes passos metodológicos: (1) uso de um verificador (publicado recentemente) para servir de base de comparação; (2) construção de dois verificadores seguindo a abordagem proposta nesta dissertação (um para comportamentos atômicos, outro para comportamentos não-atômicos); (3) síntese de erros artificiais de projeto para avaliar os verificadores; (4) simulação de diferentes representações de projetos (com e sem erros), utilizando cada um dos verificadores realizados; e (5) comparação dos verificadores construídos com o verificador-base em termos de esforço e eficácia em detectar erros (sob exatamente os mesmos parâmetros de geração).

Resultados e Discussão

Os dois novos verificadores construídos com a abordagem proposta foram comparados com um verificador-base, o qual baseia-se em múltiplas *scoreboards* relaxadas. Para a simulação de representações de projeto foi adotado o gem5, utilizando projetos de 8, 16 e 32 núcleos, com execução *out-of-order* e um protocolo MOESI de dois níveis. A arquitetura-alvo foi o ARMv8 e os verificadores foram avaliados sobre dois tipos de projetos (um com comportamentos atômicos, outro com comportamentos não-atômicos). Para a geração de testes, foi adotada uma técnica de geração dirigida disponível no *framework* utilizado, usando-se diferentes tamanhos de teste (1ki, 2ki e 4ki operações de memória). Para quantificar falsos diagnósticos foram utilizados projetos que não continham erro algum. Depois, inseriu-se nove diferentes erros artificiais para desafiar os verificadores, cada projeto contendo um único erro distinto. Para um dado tamanho de teste, o gerador foi executado doze vezes explorando diferentes sementes aleatórias, resultando em doze conjuntos de testes distintos. Foi determinada a fração de conjuntos de testes para a qual cada verificador acusou falsos diagnósticos em um projeto correto. O esforço gasto em uma tentativa para encontrar um dado erro foi medido como o tempo de execução até o erro ser encontrado ou até a geração parar, tomando a média de tempo sobre todos os conjuntos de testes. O *overhead* do verificador proposto em relação ao base foi obtido calculando a porcentagem de esforço adicional. Ao verificar projetos corretos com comportamentos não-atômicos, o verificador-base apresentou uma fração não desprezível de falsos diagnósticos. Para um dado número de núcleos, essa fração aumenta significativamente com o tamanho de teste, o que é inconveniente, porque testes maiores usualmente são necessários para expor os erros mais sutis. Para programas de 4Ki operações, o verificador-base acusou falsos positivos para um terço dos conjuntos de testes. Como resultado da modelagem apropriada do comportamento de memória, o verificador proposto para comportamentos não-atômicos não acusou qualquer falso diagnós-

tico sob as mesmas condições. Para construir um verificador adequado para comportamentos atômicos, aplicou-se requisitos adicionais a fim de restaurar a atomicidade de escrita (originalmente relaxada). Esse verificador foi comparado com o base para detectar erros em projetos contendo falhas. Ele foi capaz de encontrar todos os erros de projeto estudados, enquanto o verificador-base foi incapaz de encontrar um deles. Além disso, o máximo *overhead* de esforço observado do verificador proposto em relação ao base foi 2.5%, enquanto a máxima redução de esforço foi 16%. Os resultados indicam que a versalidade da abordagem e a melhoria na qualidade de verificação levam um custo adicional de esforço negligenciável, sendo adequada para verificar projetos com diferentes graus de relaxação da atomicidade de escrita.

Considerações Finais
Ao invés de depender da geração de testes *litmus*, cujo controle de cobertura é limita, esta dissertação aborda a verificação de memória compartilhada sob geração de testes dirigida. Ao contrário de testes *litmus*, testes aleatórios não são auto-verificáveis. Por isso, ao adotá-los, a validação de memória compartilhada requer o uso de verificadores de modelos de consistência de memória, os quais se baseiam na análise de *traces* de operações de memória monitoradas em cada núcleo. Convencionalmente, apenas a interface com a memória compartilhada é monitorada, resultando em um *único trace* de memória por núcleo. Contudo, a validação de memória compartilhada baseada em um único *trace* por núcleo é intratável. Por essa razão, verificadores convencionais exibem comportamento exponencial com números crescentes de núcleos, exceto quando sacrificam garantias de verificação. Diante disso, esta dissertação aborda verificadores usando *múltiplos* monitores por núcleo para reduzir a complexidade do problema e manter a verificação escalável. Esses monitores adicionais são inseridos na entrada e na saída de *cache buffers* e *pipeline buffers*. Consequentemente, tais monitores podem acabar expondo comportamentos especulativos e comportamentos de escrita não-atômicos resultantes de otimizações de projeto (artefatos de projeto), as quais são necessárias em implementações de alto-desempenho de uma dada arquitetura. Por essa razão, esta dissertação propôs uma abordagem para construir verificadores que são capazes de lidar com projetos apresentando efeitos especulativos e comportamentos não-atômicos. A especificação abstrata proposta é geral o suficiente para construir verificadores eficientes para projetos expondo tanto comportamentos de escrita atômicos quanto não-atômicos. A abordagem é bastante independente de arquitetura, pois captura requisitos mínimos comuns a todas as arquiteturas, e é bastante independente de *micro*arquitetura, porque o *template* de observabilidade proposto utiliza monitores localizados na interface de estruturas bem comuns. A evidência experimental indica que um verificador produzido com a abordagem é efetivo, frequentemente reduz o esforço para detectar erros e é adequado para verificar projetos com diferentes graus de relaxação da atomicidade de escrita. Portanto, esta dissertação contribuiu com a construção de um *framework* de verificação onde a geração de testes visa alta cobertura com baixo esforço, e a verificação em tempo de execução visa a descoberta de erros eficiente com suporte a diferentes arquiteturas e variantes de microarquitetura. As contribuições relacionadas a esta dissertação foram reportadas (em co-autoria): dois trabalhos em eventos internacionais e dois artigos científicos publicados em periódicos qualificados.

Palavras-chave: *Multicore*. Memória compartilhada. Verificação. Geração aleatória de testes. Consistência. Coerência. Atomicidade de escrita.

ABSTRACT

Single-chip multiprocessors, known as multicore chips, usually provide a coherent shared memory abstraction, which can be achieved by implementing a cache coherence protocol on hardware. It has been shown that on-chip hardware coherence can scale gracefully as the number of cores grows. Since scaling estimates show that the number of active cores is limited by thermal power, cache coherence can be expected to keep playing a major role for multicore chips targeting general purpose applications. Besides, multicore architectures are likely to largely relax sequential consistency constraints on store atomicity and on the ordering between loads and stores. As a result, the validation of shared memory in multicore chips faces two main challenges: the higher number of valid execution witnesses resulting from consistency relaxation and the larger coherence protocol's state space induced by growing core counts. At the microarchitectural level, the validation effort relies on simulation-based functional verification of the design representation. Litmus test generation is an effective approach to exposing consistency errors without the need for specialized checkers, but has limited coverage when used for functional verification. Constrained random test generation is an alternative approach that leads to higher coverage, but requires an independent checker to validate the observed memory behaviors against a given memory consistency model. This dissertation reports a shared memory verification framework and proposes a novel approach to contribute to one of its modules. The framework is the result of a collective research effort, and it relies on coverage-directed random test generation. The proposed approach allows the customization of efficient runtime checkers according to architecture and microarchitecture targets. It relies on an abstract specification for shared memory behavior and on an observability template for guiding the insertion of monitors in appropriate points of the design representation. To ensure scalability without compromising verification guarantees, multiple monitors per core domain are defined. As opposed to conventional checkers, which are unable to properly handle behavior arising from relaxed store atomicity, the proposed approach allows the building of checkers compliant with either relaxed or strict atomicity. We compared a checker built under the novel approach with a conventional one, both within the reported verification framework. They were evaluated when running the same test suites, each built with many programs of fixed size. For programs with 4Ki instructions, the conventional checker raised false positives for 1/3 of the test suites when targeting correct 32-core designs with relaxed atomic behavior, whereas the new checker raised none. For faulty designs with strict store atomicity, the new checker displayed negligible effort overhead when compared to the conventional one, and often led to effort reduction. Therefore, the experimental evidence indicates that a checker produced with the novel approach is effective, it often reduces the effort to detect an error, and it is suitable to checking designs with different degrees of store atomicity relaxation.

Keywords: Multicore. Shared memory. Verification. Random test generation. Consistency. Coherence. Store atomicity.

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

W	Write operation
R	Read operation
L	Load instruction
S	Store instruction
n	Number of memory operations
s	Number of shared memory locations
k	Number of distinct cache sets to which locations can be mapped
cbc	Competition biasing constraint
sbc	Sharing biasing constraint
abc	Alignment biasing constraint
χ	Maximum number of locations mapped to a same cache set
α	Cache associativity
TC	Transition coverage
TC_i	Transition coverage of Class i
O_j	j-th memory operation
p	Number of cores in a multicore chip
$(O_j)_a^i$	Memory operation O_j issued by core i to location a
$(R_j)_a^i$	Read completion event R_j on core i for location a
$(r_j)_a^i$	Read commit event r_j on core i for location a
$(W_j)_a^i$	Write completion event W_j on core i for location a
$(w_j)_a^i$	Write commit event w_j on core i for location a
$(\omega_j)_a^i$	Availability event ω_j on core i for location a
I_j	j-th memory instruction
$\prec_{po}$	Program order
$\mathscr{O}$	Set of memory operations issued by all p cores
$\mathscr{S}$	Set of store operations issued by all p cores
$\mathscr{L}$	Set of load operations issued by all p cores
$\mathscr{O}^i$	Set of memory operations issued by core i
$\mathscr{S}^i$	Set of store operations issued by core i
$\mathscr{L}^i$	Set of load operations issued by core i
$\mathscr{O}_a^i$	Set of memory operations issued by core i to location a
$\mathscr{S}_a^i$	Set of store operations issued by core i to location a
$\mathscr{L}_a^i$	Set of load operations issued by core i to location a
Val_a^0	Initial value stored at location a
$\leq$	Partial order of abstract events defining valid behaviors
$\oplus$	Monitor sampling local commit events
$\ominus$	Monitor sampling local completion events

$\odot$	Monitor sampling remote commit events
$\otimes$	Monitor sampling remote completion events
$\sigma_L(L_a^i)$	Set of stores locally observable by L_a^i
$\sigma_G(L_a^i)$	Set of stores globally observable by L_a^i
$Max[\sigma_L(L_a^i)]$	Last store locally observable by L_a^i
$Max[\sigma_G(L_a^i)]$	Last store globally observable by L_a^i
$Val[(O_j)_a^i]$	Value read or written by operation j at core i to location a
F	Memory fence
$\oslash$	Monitor sampling availability events
$\circledast$	Monitor sampling squash events
$\mathscr{E}$	Set of all physical events monitored for a given execution witness
$<$	Observed order of physical events
$\mathscr{E}^i$	Set of physical events observed in the domain of core i
$\mathscr{R}_{\ominus}^i$	Set of read completion events corresponding to stores issued by core i
$\mathscr{R}_{\oplus}^i$	Set of read commit events corresponding to stores issued by core i
$\mathscr{R}_{\circledast}^i$	Set of read squash events corresponding to stores issued by core i
$\mathscr{W}_{\oslash}^i$	Set of write availability events corresponding to stores issued by core i
$\mathscr{W}_{\oplus}^i$	Set of write commit events corresponding to stores issued by core i
$\mathscr{W}_{\ominus}^i$	Set of write completion events corresponding to stores issued by core i
$\mathscr{W}_{\circledast}^i$	Set of write squash events corresponding to stores issued by core i
$\mathscr{W}_{\odot}^i$	Set of write commit events corresponding to stores issued by cores other than i
$\mathscr{W}_{\otimes}^i$	Set of write completion events corresponding to stores issued by cores other than i
$R_{\ominus}^i$	Subset of completion events corresponding to committed loads
$\overline{R}_{\ominus}^i$	Subset of completion events corresponding to squashed loads
$W_{\oslash}^i$	Subset of availability events corresponding to committed stores
$\overline{W}_{\oslash}^i$	Subset of availability events corresponding to squashed stores

CONTENTS

1 INTRODUCTION

At the beginning of the 21st century, the end of the Dennard scaling (DENNARD et al., 1974) created a new trend in the microprocessor industry. To keep taking advantage of the continuous increase in transistor counts as predicted by Moore's Law (MOORE, 1965), manufacturers started to adopt multiprocessor designs.

A multiprocessor can be defined as a computer "consisting of tightly coupled processors whose coordination and usage are typically controlled by a single operating system and that share memory through a shared address space" (HENNESSY; PATTERSON, 2017). Nowadays, multiprocessors are ubiquitous, being present in different systems, from clusters and servers to personal computers and smartphones.

Single-chip multiprocessors, known as multicore chips, usually adopt a cache hierarchy to improve memory performance. Each core can have one or more levels of *private* caches, which introduce the problem of incoherence: different private caches may end up with distinct values for a same shared memory location. To avoid this problem, cache coherence protocols are widely adopted, providing a coherent shared memory abstraction. It has been shown that on-chip hardware coherence can scale gracefully as the number of cores grows (MARTIN; HILL; SORIN, 2012; DEVADAS, 2013). Since scaling estimates show that the number of active cores is limited by thermal power (HENNESSY; PATTERSON, 2017), we can expect, in the foreseeable future, that cache coherence will keep playing a major role for multicore chips targeting general purpose applications.

A *memory consistency model* (MCM) (ADVE; GHARACHORLOO, 1996) defines shared memory behavior for programmers and chip designers. It specifies the ordering of accesses to distinct locations (consistency rules), the ordering of stores to the same location (coherence requirement), and when a value written by a store can be observed by loads in the same or in other cores (store atomicity). The simplest memory model, sequential consistency (LAMPORT, 1979) fully enforces program ordering on memory accesses, and leads to *strict* store atomicity, that is, *multiple-copy atomic* (MCA) stores (COLLIER, 1992), which behave as if the written value was made available to all cores at the same time.

Most manufacturers have been building processors that relax sequential consistency (SC). For instance, the x86 model allows a non-conflicting load to overtake a store that precedes it in program order, and it provides *strong* store atomicity, that is, *read-own-write-early-multiple-copy atomic* (rMCA) stores (GHARACHORLOO, 1995), which behave such that the written value can be read by the issuing core before it is made available to all other cores. Some modern architectures (e.g. IBM Power, and ARMv7) largely relax program order, and provide *relaxed* store atomicity, that is, *non-multiple-copy atomic* (nMCA) stores (TRIPPEL et al., 2017), which behave such that the written value can be made available to different cores at distinct times.

1.1 SHARED MEMORY VALIDATION

Modern architectures challenge the validation of the shared memory behavior at different abstraction levels and distinct phases of the design cycle. At the architectural level, the validation effort combines formal verification (ZHANG et al., 2015) and simulation of the coherence protocol. At the microarchitectural level, it relies on simulation-based functional verification of the design representation (ADIR et al., 2004; FINE; ZIV, 2003; WAGNER; BERTACCO, 2008; QIN; MISHRA, 2012; ELVER; NAGARAJAN, 2016). Ultimately, it finishes with the test of the multicore chip prototype (HANGAL et al., 2004; MANOVIT; HANGAL, 2006; HU et al., 2012).

The dynamic validation of a multicore chip relies on a suite of parallel programs (tests), which are directly executed on the prototype or on a simulation of its design representation. The quality of dynamic validation can be tracked, for instance, by the following figures of merit: (1) *coverage*: the fraction of the design structure or the fraction of its behavior that has been validated (e.g. lines of HDL code, states, or transitions); (2) *error discovery rate*: the fraction of all tests that are able to raise an error; (3) *effort*: the amount of time spent to accomplish validation for a given test suite; and (4) *diagnosis guarantees*: to which extent diagnosis can be trusted, i.e., if apparent errors might be raised for a correct design (false positives) or if actual design errors might be overlooked (false negatives) even when they are covered.

Shared memory *test* relies on constrained *Random Test Generation* (RTG) and post-mortem checking, e.g. Manovit & Hangal (2006). It can exploit long tests with hundreds of thousands of operations to reach high coverage, because the speed of the hardware prototype allows for suitable test throughput.

Shared memory *verification* can also rely on constrained RTG when combined with runtime checking, which stops simulation as soon as a design error is found (ADIR et al., 2004; SHACHAM et al., 2008; FREITAS; RAMBO; SANTOS, 2013). However, it should exploit short tests with tens of thousands of operations (ADIR et al., 2004), because the speed of the simulator would limit test throughput if much larger tests were used. Besides, *Directed Test Generation* (DTG) has been advocated for proper coverage with smaller verification effort (WAGNER; BERTACCO, 2008; QIN; MISHRA, 2012; ELVER; NAGARAJAN, 2016).

Moreover, the exploitation of MCM checkers for validation of shared memory behavior, originally proposed for offline post-silicon test (HANGAL et al., 2004), has been adopted for building runtime checkers (SHACHAM et al., 2008; FREITAS; RAMBO; SANTOS, 2013) suitable to the pre-silicon verification phase.

Litmus test generation combined with checking of legal execution witnesses constitutes an alternative approach to shared memory validation, and can be used for both, verification and test, e.g. Alglave, Maranget & Tautschnig (2014), and Lustig et al. (2017). It exploits an MCM to synthesize small test programs that are able to expose invalid execution witnesses. Albeit quite effective to find errors, they have limited coverage at design time (ELVER; NAGARAJAN, 2016).

Since coverage is crucial for verification closure (FINE; FOURNIER; ZIV, 2009), this dissertation addresses the combination of test generation and MCM checkers in the scope of the *functional verification* of coherent shared memory behavior at design time.

1.2 VERIFICATION CHALLENGES

It has been shown that multicore scaling is power limited (ESMAEILZADEH et al., 2011; ESMAEILZADEH et al., 2012), as a result of the under-utilization of integration capacity (a.k.a. dark silicon). In spite of that, the projected scaling still leads to a major verification challenge, because the protocol state space grows exponentially with the number of cores. Therefore, pragmatic techniques should not enumerate the state space. Besides, conventional checkers with limited observability (one monitor per core) trade verification guarantees for effort scalability (and vice-versa). That is why checkers targeting shared memory verification should take advantage of the higher observability of design representations, and use multiple monitors per core, providing effort scalability without compromising verification guarantees.

The relaxation of order and store atomicity increases the number of valid execution witnesses as compared to SC, making it harder for functional verification to *expose* invalid ones. Besides, the relaxation of store atomicity makes it harder to *tell* a valid execution witness from an invalid one, because conventional checkers assume rMCA stores (MANOVIT; HANGAL, 2006; SHACHAM et al., 2008; HU et al., 2012; FREITAS; RAMBO; SANTOS, 2013). Some architectures have been revised to disallow nMCA behavior, such as ARMv8 (PULTE et al., 2017) and RISC-V. However, high-performance implementations may still exhibit non-atomic behavior due to design optimizations (e.g. speculation and early acknowledgements), as long as it is not exposed to the programmer (WATERMAN; ASANOVI, 2019). In other words, non-atomic behavior is allowed as far as it is *not* architecturally visible. In such implementations, memory model checkers exploiting higher observability for runtime efficiency may still end up exposing non-atomic behavior.

When targeting a design with nMCA behavior, a conventional rMCA checker is bound to raise false positives, because rMCA is a stronger condition than nMCA, whereas an nMCA checker is effective in detecting actual errors for a faulty design. However, when targeting a design with rMCA behavior, an nMCA checker may be less effective in raising actual error diagnoses as compared to an rMCA checker, because some invalid behavior under rMCA may be valid under nMCA. Thus, the building of a checker requires an approach able to handle either rMCA or nMCA implementations of a same architecture.

1.3 GENERAL GOALS AND SPECIFIC CONTRIBUTIONS

This dissertation presents part of a collective research effort towards two ultimate goals: (1) the development of novel test generators and (2) the development of new MCM checkers, all integrated into a complete verification framework. Since test generators are required to evaluate MCM checkers, it briefly reports technical contributions to the first goal, which are part of the verification framework. It also details a collective scientific contribution to the second goal, which is the main focus of the dissertation.

The main contribution described in this dissertation is a follow up of a preliminary work, which was co-developed by Olav P. Henschel and Luiz C. V. dos Santos in 2013, but remained unpublished. It consisted of a prototype for a checker tied to an nMCA microarchitecture. Its implementation was incomplete, and it deserved experimental validation. That preliminary work was revised, extended, validated, and published in co-authorship (GRAF et al., 2019). The involvement of the author in this contribution includes (1) the redesign of the algorithms for checking the axioms, (2) the extension of the observability template to properly capture all required physical events, and (3) the change of focus to match current architectural trends that simplify store atomicity (PULTE et al., 2017).

1.4 ORGANIZATION OF THIS DISSERTATION

The remainder of this dissertation is organized as follows. Chapter 2 explains some fundamental concepts for understanding the key ideas behind the described techniques. Chapter 3 reviews related work. Chapter 4 introduces the verification framework, briefly reporting the co-developed modules. Chapter 5 describes the main scientific contribution: an approach called Spec&Check for the building of MCM-based runtime checkers. Chapter 6 experimentally evaluates checkers built under the proposed approach as compared to a conventional one. Chapter 7 puts our conclusions in perspective.

1.5 PUBLICATIONS

A significant part of the text of this dissertation reflects the contents of other documents written in co-authorship, which were the result of strongly collaborative research and were published (or accepted for publication) in the proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD) and on the journal IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems (TCAD), as follows:

- ICCAD 2019 paper on runtime checking (GRAF et al., 2019);
- ICCAD 2018 paper on DTG (ANDRADE et al., 2018);
- TCAD 2020 article on RTG (ANDRADE; GRAF; SANTOS, 2020);
- TCAD 2020 article on DTG (ANDRADE et al., 2020).

The author acknowledges the contributions of the co-authors of those documents for the joint research effort that enabled the development of his own contribution.

2 FUNDAMENTAL CONCEPTS

This chapter introduces the main notions required for understanding the techniques addressed in the remainder of this dissertation. It focuses on the concepts captured by an MCM: the coherence requirement (Section 2.1), the consistency rules (Section 2.2), and the write atomicity (Section 2.3).

2.1 CACHE COHERENCE

The introduction of private caches in multicore chips raises the problem illustrated in Table 2 for two cores, C_1 and C_2, and a shared memory location X. Since each core views the shared memory through its individual cache, two distinct cores could end up seeing two different values for the same shared location. First, let us assume the memory location X contains the value 1 (time 0). Then both cores issue load instructions that read location X, loading the value 1 to its respective private caches (times 1 and 2). Finally, core C_1 issues a store instruction, which writes a new value 0 into a local copy of X (time 3). For simplicity, we assume a write-through cache: every time a new value is written to the cache, it is also propagated to lower hierarchical levels (i.e., other cache levels and the main memory). That is why both, the cache of core C_1 and the memory, are shown with the updated value. However, the cache of core C_2 remains with the old value of X, leaving the memory in an incoherent state. If a new load is issued by core C_2, it will return 1 instead of 0, which is the most recent value.

Table 2 – The cache coherence problem.

Time	Event	Cache contents for core C_1	Cache contents for core C_2	Memory contents for location X
0				1
1	C_1 reads X	1		1
2	C_2 reads X	1	1	1
3	C_1 writes 0 into X	0	1	0

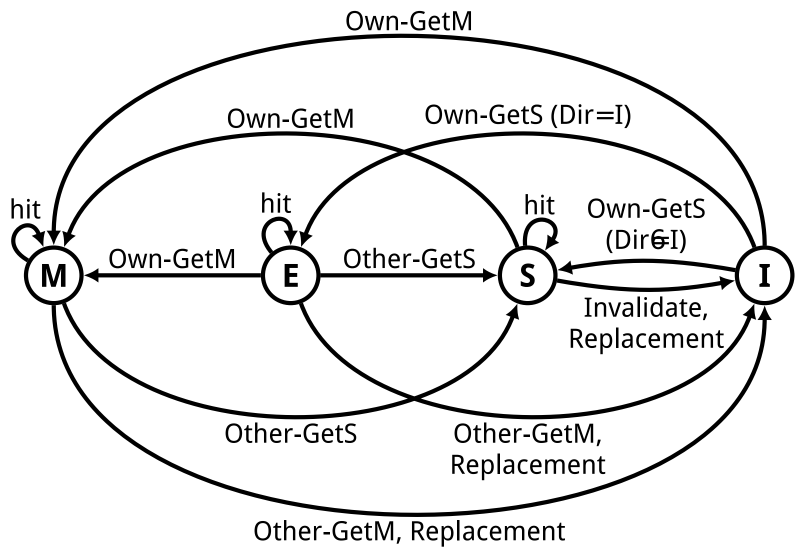
Source: adapted from Hennessy & Patterson (2017)

A memory system is considered coherent if it has three properties (HENNESSY; PATTERSON, 2017): (1) *local observability*: a read by core C_1 to location X that follows a write by the same core to X, with no other cores writing to X between them, always returns the value written by C_1; (2) *global observability*: a read by core C_1 to location X that follows a write by core C_2 returns the written value if both accesses are sufficiently apart in time and no other writes to X occur between them; and (3) *write serialization*: any two writes to the same location by any two cores are seen in the same order by all cores.

The first property preserves program order, while the second property forbids a core to continuously read an old value. The third property ensures that all writes will eventually be seen by every core. For instance, imagine two writes to the same location issued by two distinct cores, C_1 and C_2, in that order. If another core saw the inverse order, i.e., first the write by C_2 and then the write by C_1, all subsequent reads will return the value written by C_1. Therefore, the write of C_2 would be lost to this core forever. The serialization of all writes to the same location prevents this issue.

To enforce these three properties and provide a coherent memory abstraction, cache coherence protocols are usually implemented in hardware. The coherence protocol can be specified by a finite state machine (FSM) with output actions at each transition. The states of the FSM are associated with each *memory* block. The cache controllers use the FSM to manage and arbitrate the communication between distinct caches and different hierarchical levels. Figure 1 shows a simplified representation of the FSM of a protocol called MESI. The output actions at each transition are ommited for simplicity.

Figure 1 – Simplified FSM for the MESI protocol.



Source: adapted from Andrade, Graf & Santos (2020)

In the MESI protocol there are four states, Modified (M), Exclusive (E), Shared (S), and Invalid (I). Each state determines the access permissions of a memory block in a given cache, as follows:

- Invalid: neither read nor write permissions (the memory block is not present in the cache).
- Shared: read-only permission (the memory block is present in more than one private cache).
- Exclusive: read-only permission (the memory block is present only in a given cache).
- Modified: read/write permissions (the memory block is present only in a given cache).

Besides, the states encode if a block is clean or dirty. A block is clean if it has not been modified since its allocation (it has the same contents as the main memory's). A block is dirty if it has been modified and its contents are not yet updated in the main memory. In the states I, S, and E, the block is clean, in the state M, it is dirty. This allow the use of write-back caches, where the main memory is only updated when a dirty block is about to change to state I (or S), instead of updating it every time a new write happens.

The input events illustrated in Figure 1 starting with the *Own* prefix are issued by the local core (who owns the cache), while the events prefixed with *Other* are triggered by coherence messages received from private caches of remote cores. A *GetM* event represents a request to write into the memory block, while a *GetS* represents a request to read from the block. A hit event represents a successful read (or write if in state M) to the block. *Invalidate* events are the result of a *Own-GetM* event in a remote cache, while *Replacement* events are the result of a *Own-GetM* event in the local cache, but for a different memory block that happens to map to the same cache set.

2.2 CONSISTENCY RULES

Cache coherence defines a global order of memory accesses to a *single* location, but says nothing about how the order between accesses to *different* locations must be. That is why, beyond the coherence requirement, an MCM must also define consistency rules to specify which orders can be relaxed and which must be preserved. To illustrate this notion, let us start with an example.

Figure 2 shows a small parallel program with two threads of execution accessing two shared locations, A and B. Each thread is executed in a distinct core, C_1 and C_2, and the identifiers $r1$ and $r2$ denote local registers. At a first glance, it would seem that there are only three possible outcomes to the values loaded into the local registers: $r1 = 1$ and $r2 = 0$, $r1 = 0$ and $r2 = 1$ or $r1 = 1$ and $r2 = 1$. This is because we commonly assume the *program order* is always followed, but that is not always the case.

Figure 2 – Example of parallel program whose result depends on the underlying MCM.

C_1	C_2
B = 0;	A = 0;
A = 1;	B = 1;
$r1$ = B;	$r2$ = A;

Source: adapted from Hennessy & Patterson (2017)

Assume this program is being executed in a multicore which adopts the coherence protocol described in the previous section. After the first write of each thread is executed, the memory blocks containing the locations A and B are owned (state M) by the private caches of the cores C_2 and C_1, respectively. When the subsequent writes, $A = 1$ and $B = 1$, are issued to the respective caches, the write to A will invalidate the cache block in core C_2 due to an Other-GetM event, while the write to B will do the same to the block in C_1. If the cores are allowed to continue their executions before each write completes, it is possible that both reads, $r1 = B$ and $r2 = A$, see the old cached values before the outstanding invalidations (resulted from the previous writes) take effect. In that case, the final outcome would be $r1 = 0$ and $r2 = 0$. This is only possible if the order write $\rightarrow$ read (involving distinct locations) is relaxed.

The only MCM which completely preserves program order is *sequential consistency* (SC). It explicitly requires that the memory accesses of each core are kept in program order, while accesses from different cores can be arbitrarily interleaved. Albeit it makes it easier to reason about parallel programs and prevents non-obvious outcomes like $r1 = 0$ and $r2 = 0$ in the Figure 2, it also limits the range of optimizations that can be exploited by the hardware. That is why modern architectures tend to adopt relaxed consistency rules (e.g. IBM POWER9, ARMv8, and RISC-V). Table 3 compares the relaxations in program order allowed by distinct architectures, where R and W stands for read and write, respectively, and $\rightarrow$ stands for program order between operations to distinct locations.

Table 3 – Relaxations of program order allowed on different MCMs

	Relaxed orders			
MCM	$W \rightarrow R$	$W \rightarrow W$	$R \rightarrow W$	$R \rightarrow R$
x86/TSO	✓			
RISC-V	✓	✓	✓	✓
ARMv8	✓	✓	✓	✓
IBM POWER9	✓	✓	✓	✓

Source: adapted from Gharachorloo (1995)

Notice that in the previous example the outcome $r1 = 0$ and $r2 = 0$ would be allowed even by the least relaxed architecture, the x86. This is because it adopts an MCM where the execution of a read can be anticipated with respect to preceding non-conflicting writes[1], which is exactly the scenario proposed in the example.

To allow the programmer to enforce the program order, architectures implementing relaxed MCMs provide one or more instructions implementing the concept of memory fences. A *memory fence* ensures that all memory accesses preceding it in program order have finished before any subsequent memory access is executed.

2.3 STORE ATOMICITY

In a memory system with private caches, store operations are not atomic, since propagating changes to multiple caches is inherently a non-atomic operation (GHARACHORLOO, 1995). Store atomicity admits distinct degrees of relaxation. A store exhibits *multiple-copy atomic* (MCA) behavior when it appears as if atomic, i.e., a store can only be considered com-

[1] Such consistency model is known as *total store ordering* (TSO).

pleted when its value becomes visible to all cores and no loads are issued to the same location before that happens. A store exhibits *read-own-write-early-multiple-copy atomic* (rMCA) behavior, when it exhibits MCA behavior for all cores, except that the core that issued the store is allowed to read its value before the operation completes with respect to the others. Finally, a store exhibits *non-multiple-copy atomic* (nMCA) behavior, when it becomes visible to different cores (possibly) at distinct times.

To illustrate the impact of an nMCA behavior, consider the concurrent program illustrated in Figure 3. We assume that A and B are shared locations previously initialized with the value 0. The identifiers $r1$, $r2$, $r3$ and $r4$ denote local registers. Cores C_1 and C_2 perform stores S_1 and S_2, respectively. Their values might be observed by loads L_1 and L_2 in C_3, and by loads L_3 and L_4 in C_4. Suppose an execution witness where L_1 observes S_1 in C_3, resulting in $r1 = 1$, and where L_3 observes S_2 in C_4, resulting in $r3 = 1$. If it turns out that L_2 has not observed S_2 in C_3, resulting in $r2 = 0$, and L_4 has not observed S_1 in C_4, resulting in $r4 = 0$, the former would imply that C_3 sees S_1 before S_2, while the latter would imply that C_4 sees S_2 before S_1. Thus, in such case, no order would exist for the stores. Such unacceptable behavior never happens to rMCA stores for which a global linear order of stores is guaranteed. However, nMCA stores cannot guarantee such proper behavior. For this reason, architectures that allow nMCA behavior visible to the programmer, such as POWER9 (IBM, 2019), implement another type of memory fences called *cumulative* fences, which are able to restore a global order of stores when needed.

Figure 3 – Independent-read, independent-write (IRIW) example.

C_1	C_2	C_3	C_4
S_1: A = 1;	S_2: B = 1;	L_1: $r1$ = A; F_1: fence; L_2: $r2$ = B;	L_3: $r3$ = B; F_2: fence; L_4: $r4$ = A;

Source: adapted from Graf et al. (2019)

This dissertation does not discuss cumulative fences, because it focuses on the trend towards *architecturally invisible* nMCA store behavior, as observed for popular and promising architectures[2], such as ARMv8 (PULTE et al., 2017) and RISC-V (WATERMAN; ASANOVI, 2019). Recall that, despite such architecture simplification, nMCA behavior can be exposed at the implementation level.

The main concepts reviewed in this chapter pave the way towards the discussion of the next chapter, which addresses features and limitations of related works on both test generation and MCM checking techniques.

2 To the best of our knowledge, only the IBM Power architecture is not currently following that trend.

3 RELATED WORK

This chapter summarizes related work on test generators and checkers. There are two main approaches to shared memory validation: (1) litmus test generation combined with self-checking of legal execution witnesses, and (2) constrained random test generation combined with MCM checking. Section 3.1 overviews works following the first approach, while Sections 3.2 and 3.3 focus on the second approach, describing different generators and MCM checkers.

3.1 LITMUS TEST GENERATION

Litmus tests are short concurrent programs designed to stress certain MCM behaviors (as illustrated by Figure 3 in Chapter 2). The automated generation of litmus tests have been proposed (ALGLAVE et al., 2010; ALGLAVE et al., 2015; LUSTIG et al., 2017) for validating multicore chips. The MCM declares which test outcomes are legal and which are not (LUSTIG et al., 2017). Each test is run thousands of times to provoke the behavior that the test characterizes (ALGLAVE et al., 2015). Despite its success in finding subtle bugs when testing commercial chips, this approach leads to limited coverage at design time (ELVER; NAGARAJAN, 2016). Besides, when a litmus test finds an error, it does not directly indicate where the error lies in the microarchitecture (i.e., no support is offered for design debugging).

Tricheck (TRIPPEL et al., 2017) is a full-stack approach for verifying whether the language, the compiler, the ISA, and the implementation collectively satisfy MCM requirements. It was able to uncover under-specifications and potential inefficiencies in the RISC-V ISA. It does handle nMCA stores, but its underlying microarchitecture checker is based on litmus tests, thereby limiting its coverage potential.

3.2 RANDOM AND DIRECTED TEST GENERATION

Random test generation (RTG) and *directed test generation* (DTG) can lead to higher coverage than litmus tests. Random tests are concurrent programs with multiple threads, usually one per core, composed of several memory operations (mostly loads and stores) on shared locations. Non-synchronized programs are favored, since intensive data races expose bugs faster (HANGAL et al., 2004; SHACHAM et al., 2008). Industrial environments have been relying on RTG for processor verification since the mid-1980's. For instance, IBM's Genesys-Pro casts random test generation into a constraint satisfaction problem (ADIR et al., 2004).

However, as opposed to post-silicon testing, pre-silicon verification cannot afford long tests to achieve coverage goals. To reach similar goals with shorter tests, DTG has been advocated (WAGNER; BERTACCO, 2008; QIN; MISHRA, 2012; ELVER; NAGARAJAN, 2016).

In MCjammer (WAGNER; BERTACCO, 2008), each core is assigned an agent, which sees the coherence protocol in terms of a dichotomic FSM (comprising only the states of the local node and the state of the environment). The agents exploit the insufficiently verified transitions to formulate their goals towards higher transition coverage. The generator is reusable only for derivative designs that comply with a same protocol, because the FSM must be modified for each protocol variant.

Another technique (QIN; MISHRA, 2012) does not require RTG. It decomposes the entire protocol space into simpler structures such as hyper-cubes and cliques, which can be traversed in an Euler tour to avoid revisiting transitions. As a result, no transition is ever visited more than once, and tests leading to full transition coverage can be generated. However, it does not scale well with growing core counts. It deserved a recent extension (LYU et al., 2019), based on a symmetry reduction technique, which defines equivalent classes and restricts the state space to class representatives, allowing a trade-off between coverage and verification effort. However, it is not suitable for memory consistency verification, because it requires artificial order constraints[1] for proper controllability, thereby inhibiting data races. Besides, the approach leads to abstract transitions that may aggregate multiple paths over transient states, but only covers one of them in the Euler tour, limiting error discovery.

McVerSi (ELVER; NAGARAJAN, 2016) proposes a Genetic Programming approach to DTG, where an RTG technique is used only for the creation of an initial population of tests. Further generations of tests are obtained from a pre-existing population by using as objective function the fitness of a test, which is obtained from some coverage metric defined by the verification environment. To obtain a new population from the fittest tests, it employs a selective crossover function that favors the selection of memory operations contributing to higher non-determinism.

Although McVerSi reaches higher coverage and error discovery rate when compared to litmus test generation and pure RTG, a more recent work (ANDRADE et al., 2018) introduces a novel technique with steeper coverage evolution and superior discovery rate. It internally employs two other (RTG) techniques (ANDRADE; GRAF; SANTOS, 2016; ANDRADE; GRAF; SANTOS, 2020). Instead of relying only on target program parameters (e.g. number of operations and locations) as constraints, they allow the exploitation of consistency rules and general properties of coherence protocols and cache memories as extra constraints for improved RTG. Chapter 4 illustrates the main ideas behind these techniques, since they are part of the framework under construction.

3.3 MCM CHECKERS

Memory model checkers exploit the MCM for reducing the coupling between testing and implementation details. Ideally, the following *correctness problem* (CHATTERJEE; SIVARAJ; GOPALAKRISHNAN, 2002) should be addressed by an MCM checker: given a memory model specification and an implementation of a multiprocessing system, check if *all* the executions generated by the latter satisfy the former for *any* given parallel program.

[1] It employs thread barriers to create a global serialization of all instructions.

On the one hand, formal approaches (PARK; DILL, 1995; HENZINGER; QADEER; RAJAMANI, 1999; CHATTERJEE; SIVARAJ; GOPALAKRISHNAN, 2002; ABTS; SCOTT; LILJA, 2003; GOPALAKRISHNAN; YANG; SIVARAJ, 2004) address simplified instances of that problem by checking the correctness of an intermediate abstraction of the actual implementation against the memory model specification. Although such approaches can find bugs arising from early phases of the design (e.g. protocol bugs), they miss errors that are present only in the implementation.

On the other hand, dynamic methods can check executions that exercise the system in all its detail: the actual hardware or a prototype of it (BARROSO et al., 1995; HANGAL et al., 2004; MANOVIT; HANGAL, 2005; ROY et al., 2006; MANOVIT; HANGAL, 2006; CHEN; MALIK; PATRA, 2008; DEORIO; WAGNER; BERTACCO, 2009; CHEN et al., 2009; HU et al., 2012; MAMMO et al., 2015; LEE; BERTACCO, 2017), or its simulated design representation (SHACHAM et al., 2008; RAMBO; HENSCHEL; SANTOS, 2012; FREITAS; RAMBO; SANTOS, 2013). However, they solve instances of the correctness problem that are limited to *one* execution of *a* given parallel program (i.e., a test), albeit several instances are solved to improve the chances of exposing errors (i.e., a test suite).

Most checkers are tied to a given design phase. Among the checkers tailored to post-silicon usage, some require hardware changes in the memory subsystem (CHEN; MALIK; PATRA, 2008; DEORIO; WAGNER; BERTACCO, 2009; MAMMO et al., 2015). For instance, Dacota (DEORIO; WAGNER; BERTACCO, 2009) modifies the memory hierarchy and the interconnect to observe read mapping[2] and total write order[3], which leads to linear-time checking. However, some industrial environments may not envisage the use of dedicated hardware for memory model verification. That is why many post-silicon checkers are entirely based on software (HANGAL et al., 2004; ROY et al., 2006; MANOVIT; HANGAL, 2005; MANOVIT; HANGAL, 2006; HU et al., 2012; LEE; BERTACCO, 2017).

Since we focus on dynamic checkers for pre-silicon verification, formal approaches and techniques requiring hardware assistance are outside the scope of this dissertation. As the post-silicon checkers not requiring hardware assistance are may be used at pre-silicon time, they are included in our discussion. Therefore, from now on, we target only the software-based approaches, which can be split into two categories, as follows.

Postmortem checkers (HANGAL et al., 2004; ROY et al., 2006; MANOVIT; HANGAL, 2005; MANOVIT; HANGAL, 2006; CHEN et al., 2009; HU et al., 2012; RAMBO; HENSCHEL; SANTOS, 2012; LEE; BERTACCO, 2017) are launched after the execution of a parallel program. Only then the traces (or behaviors) produced by the execution are actually checked. In contrast, *runtime checkers* (SHACHAM et al., 2008; FREITAS; RAMBO; SANTOS, 2013) are launched simultaneously with program execution. Traces are incrementally checked as they are produced and program execution is aborted as soon as an error is found.

The next subsections present a detailed review of postmortem and runtime checkers,

[2] It maps each read operation to the write operation producing its value.
[3] It specifies the ordering of write operations to the same location.

Table 4 – A comparison of dynamic memory model checkers.

Technique	Category	Store Atomicity	Guarantees	Monitors	Scalable (w.r.t. core count)
(HANGAL et al., 2004)	postmortem	strong	none	1	yes
(MANOVIT; HANGAL, 2005)	postmortem	strong	none	1	yes
(ROY et al., 2006)	postmortem	strong	none	1	yes
(MANOVIT; HANGAL, 2006)	postmortem	strong	proven	1	no
(CHEN et al., 2009)	postmortem	strong	proven	1	no
(HU et al., 2012)	postmortem	strong	proven	1	no
(RAMBO; HENSCHEL; SANTOS, 2012)	postmortem	strong	proven	2	no
(LEE; BERTACCO, 2017)	postmortem	strong	none	1	yes
(SHACHAM et al., 2008)	runtime	strong	none	1	yes
(FREITAS; RAMBO; SANTOS, 2013)	runtime	strong	proven	3	yes
This work	runtime	strong or relaxed	[4]	6	yes

Source: the author.

as summarized in Table 4. The table indicates the type of store atomicity supported by each checker and whether it offers or not proven guarantees of finding an error exposed by a given test. It also shows the required observability, expressed as the number of monitors per core, and whether it is scalable or not with respect to growing core counts. The table's last row contrasts the approach proposed in this dissertation with the related works.

3.3.1 Postmortem checkers

Most postmortem checkers are based on inferences (HANGAL et al., 2004; ROY et al., 2006; MANOVIT; HANGAL, 2005; MANOVIT; HANGAL, 2006; HU et al., 2012; LEE; BERTACCO, 2017), although a method based on bipartite graph matching is also reported (RAMBO; HENSCHEL; SANTOS, 2012).

Inference-based checkers solve instances of the verification problem by encoding properties and axioms of the memory model into a *constraint graph*. They make successive inferences on the order of operations by inserting new edges in the constraint graph until a cycle is detected or no more inferences can be made. Since only a directed *acyclic* graph (DAG) can represent an order relation, the detection of a cycle in the constraint graph is a proof of a violation of the memory model. The fact that no cycles are detected, however, is not a proof of valid behavior, because some existing order relation between operations might not have been inferred (MANOVIT; HANGAL, 2006). Inference-based checkers were originally designed for the limited observability that constrains post-silicon verification (one monitor per core). However, they are often used as pre-silicon checkers.

The use of a constraint graph for memory model checking was first proposed in TSOtool (HANGAL et al., 2004). Several elaborations on that basic mechanism were reported.

[4] The verification guarantees of the proposed approach were not determined by the time of writing, and they will be addressed as future work. However, the approach's formal basis and its higher observability make it promising to well-established guarantees.

A refinement based on more efficient heuristics (MANOVIT; HANGAL, 2005) reduced the worst-case complexity. A more comprehensive technique (ROY et al., 2006) generalized the key ideas from (HANGAL et al., 2004) to handle processors implementing multiple memory models. The technique's execution time was improved by using incremental graph closure and parallelization. However, these early tools (HANGAL et al., 2004; MANOVIT; HANGAL, 2005; ROY et al., 2006) may raise false negatives. They were designed as best effort checkers with no verification guarantees whatsoever.

To rule out false negatives when analyzing a given set of traces, a few checkers rely on backtracking (MANOVIT; HANGAL, 2006; CHEN et al., 2009; HU et al., 2012), leading to higher computational effort as compared to best effort checkers. As a result, test effectiveness is improved at the expense of worst-case time complexity and average runtime. The use of backtracking was first proposed in (MANOVIT; HANGAL, 2006). Then LCHECK (CHEN et al., 2009) improved backtracking by relying on the notion of *pending period*, which is the interval containing the time when an operation is considered to be globally performed (i.e. committed with respect to all cores). A time order induced by pending periods is used to prune the verification space. The technique requires trivial hardware instrumentation (two hardware registers per core and an additional instruction to observe each load value). Later, XCHECK (HU et al., 2012) eliminated the need for hardware instrumentation and further exploited the pending periods with *reusable cycle checking*, a technique that bounds the number of inferences when incrementally checking for a violation (cycle). As a result, the analysis can be performed in linear time with the number of operations. However, its worst-case complexity still has an exponential component with respect to the number of cores.

A recent work (LEE; BERTACCO, 2017) proposes a novel, minimally intrusive code-instrumentation technique to generate the traces of memory accesses. Given a constrained-random test, the technique augments the test code to enable the computation of per-thread signatures. A signature represents the result of the execution of the loads in a thread. Then, all per-thread signatures are concatenated into a single execution signature. Each unique value of the execution signature represents one possible execution witness of the test, and can be used to reconstruct the trace of memory accesses and build the corresponding constraint graph. Besides, the work also proposes the exploitation of similarities among constraint graphs resulting from distinct executions of the same test to accomplish a collective graph checking. However, since it does not employ backtracking and is limited to one trace per core, the technique does not provide verification guarantees (it may raise false negatives).

The only postmortem checker not based on inferences was proposed by Rambo, Henschel & Santos (2012). It offers similar guarantees as previous works (MANOVIT; HANGAL, 2006; CHEN et al., 2009; HU et al., 2012) without the need for backtracking. It relies on sampling two sequences of memory events per core. Since it reuses "as is" the (extended) matching algorithm proposed in (MARCILIO et al., 2009), which solved a *more general* problem, it unnecessarily inherits a high worst-case complexity, which could have been reduced by tailoring the matching algorithm to the actual target instance. The experimental evidence show, however, that the average computational effort is much smaller when using RTG, but it is still higher when compared to runtime checkers (FREITAS; RAMBO; SANTOS, 2013).

3.3.2 Runtime checkers

The use of a relaxed scoreboard was proposed for fast runtime checking (SHACHAM et al., 2008). As opposed to a conventional scoreboard, which admits a single event per entry, the relaxed scoreboard keeps multiple expected events per entry when a single memory event cannot be deterministically identified. It employs an update rule that stores a new event after each write operation and dynamically removes events that become invalid after each read operation. Since it never reconsiders a previous decision and assumes a single monitor per core, the technique admittedly may raise false negatives for a given test program.

In contrast, another work (FREITAS; RAMBO; SANTOS, 2013) proposes the use of multiple relaxed scoreboards (one per core) to build a runtime checker. An error is raised as soon as it is locally found by a relaxed scoreboard or by a global checker. When three monitors are used per core, the checker offers proven guarantees: neither false negatives nor false positives are ever raised for memory models not requiring total store ordering. Due to the higher observability at design time, it is scalable, and its multiple monitors have the advantage of better locating where the error lies, which eases debugging.

3.3.3 Discussion

Most verification approaches (HANGAL et al., 2004; MANOVIT; HANGAL, 2005; MANOVIT; HANGAL, 2006; ROY et al., 2006; CHEN et al., 2009; RAMBO; HENSCHEL; SANTOS, 2012; FREITAS; RAMBO; SANTOS, 2013) rely on automatically generating lock-free parallel programs synthesized with RTG. Such synthetic programs have one thread per core, each executing a sequence of operations that make references to shared locations. Both the sequence and the references are generated pseudo-randomly as a way to improving error coverage. Such approaches constrain test generation so that unique values are written to different addresses, i.e. the read mapping is known by construction. Since XCHECK (HU et al., 2012) does not require such enforcement (it indeed 'recovers' the read mapping from the pending periods), it can handle ordinary programs. However, it is unclear whether or not pre-silicon verification could actually benefit from ordinary programs at early design phases.

DTG can reduce the verification effort as compared to RTG because the former wastes less time in simulating tests that would redundantly stimulate the same state transitions. However, DTG neither can reduce the effort resulting from false negatives (when checkers lacking verification guarantees overlook errors whose early detection would save simulation time) nor it can reduce the effort due to limited observability (when checkers require backtracking to provide verification guarantees). Among the methods with proven guarantees, the work by Manovit & Hangal (2006) is the one with the poorest scalability with core count. Although LCHECK and

XCHECK are more scalable, the exponential component in their worst-case complexity admittedly limits their long-term scalability, since it may exponentially affect the average runtime. In contrast, the higher observability at design time exploited in Rambo, Henschel & Santos (2012) and Freitas, Rambo & Santos (2013) results in scalable checkers without compromising verification guarantees.

Both LCHECK and XCHECK rely on the notion of pending period, which requires strong store atomicity. Therefore, they cannot provide guarantees under relaxed atomicity, i.e., when nMCA store behavior is visible at the architectural level. In this case, they may raise false positives, despite their expensive backtracking mechanism. However, LCHECK and XCHECK can handle designs where nMCA behavior is exposed at the implementation level only. On the one hand, it is unclear if Manovit & Hangal (2006) could be generalized for relaxed MCMs while preserving guarantees. Its extremely poor scalability would anyway hamper its practical use as a pre-silicon checker. On the other hand, the checkers reported in Rambo, Henschel & Santos (2012) and Freitas, Rambo & Santos (2013) employ multiple engines for locally checking each core individually and a single engine to perform global checking. They employ different types of local engines, but both types support read forwarding and each engine sees a single write event for each store instruction. Therefore, the local checkers do not impair the verification of relaxed models. However, they share a global engine that requires strong store atomicity. Therefore, neither of them can preserve their proven guarantees against designs with nMCA behavior.

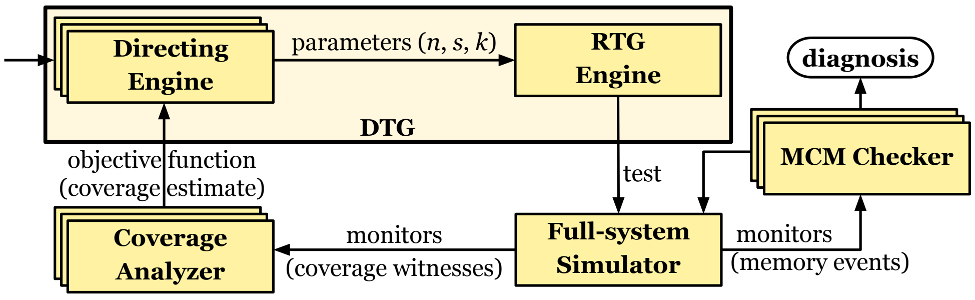
Thus, to face modern architectures, growing core counts, coverage goals, and effort requirements at design time, a checker should: (1) handle nMCA stores (to rule out false error diagnoses), (2) rely on multiple monitors (to be scalable), (3) comply with RTG or DTG (to avoid limiting coverage), and (4) stop simulation as soon as an error is hit (to reduce effort).

Chapter 5 will eventually show how to specify and build checkers with such features. However, to better explain the impact of test generators on MCM checking, the next chapter presents an overview of the verification framework in which the checkers lie. This is especially important for the experimental evaluation to be described in Chapter 6.

4 THE VERIFICATION FRAMEWORK

The framework under construction can be split into different cooperating engines, as depicted in Figure 4. It relies on coverage-driven DTG combined with runtime MCM checking for the functional verification of shared memory behavior. Its generation engines were designed to be reusable across derivative designs, different protocol variants, and distinct coverage metrics. The RTG engine relies on three parameters to constrain the building of a test program: the number of memory operations (n), the number of shared memory locations (s), and the number of distinct cache sets to which the locations can be mapped (k). While the simulator executes a test program, monitors observe memory events at relevant points of each core domain. A checker analyzes the monitored events at runtime according to the axioms of the target MCM. Besides, other monitors observe events that serve as coverage witnesses from which a coverage analyzer computes the cumulative coverage of all tests executed so far. The directing engine takes that coverage value into account before selecting the next setting of parameters for RTG.

Figure 4 – An overview of the framework under construction.



Source: adapted from Andrade et al. (2018)

To provide a full-system design representation of a multicore chip, we adopted the popular gem5 simulator (BINKERT et al., 2011), which provide many alternatives for the choice of ISA, (microarchitectural) CPU model, coherence protocol, and cache parameters.

The framework admits multiple checkers, depending on the target MCM, and it allows the experimentation with distinct techniques, as far as they follow a standard interface to communicate with the artifacts required for monitor insertion[1].

Similarly, multiple coverage analyzers can be built, one for each different coverage metric that might be adopted. They represent part of the technical contributions to the co-developed engines.

Most of the technical contributions lie in the RTG engine (ANDRADE; GRAF; SANTOS, 2020) and in the directing engine (ANDRADE et al., 2018), whose main ideas are summarized in the next sections.

[1] For instance, the checkers proposed in Chapter 5, as well as the checker used as a baseline in Chapter 6 (FREITAS; RAMBO; SANTOS, 2013), can all be used interchangeably, because they were implemented according to that standard interface.

4.1 RTG ENGINE

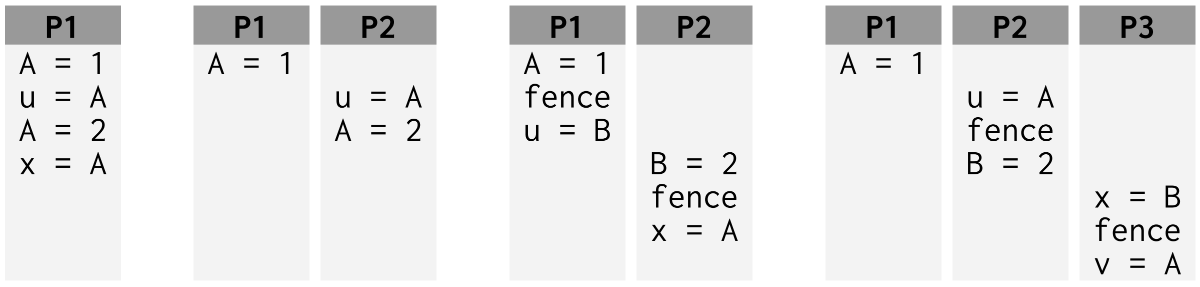
RTG is decomposed into two main sub-problems: (1) constrained random thread generation and (2) constrained random address assignment. The motivation for such decomposition lies in the fact that short random tests are unlikely to induce enough racy operations and sufficient eviction events for adequate coverage unless generation sub-tasks are properly constrained. The decomposition results from two properties of canonical dependence chains (GHARACHORLOO, 1995): their ability to favor orderings leading to coherence events and their independence from the effective addresses assigned to the shared locations.

The novel techniques proposed in (ANDRADE; GRAF; SANTOS, 2020) exploit non-conventional constraints on RTG. The reduction in scope and the decomposition fostered the design of novel, specific algorithms[2] to solve the sub-problems instead of relying on generic solvers. The RTG engine embeds two non-conventional techniques: (1) *chaining*, which exploits multiprocessor dependence chains for constraining thread generation; and (2) *biasing*, which exploits a partitioning of the shared locations for constraining their effective addresses. The next sections provide an overview of these techniques by means of illustrative examples.

4.1.1 Thread generation: chaining

The main idea behind thread generation is the exploitation of uniprocessor and multiprocessor dependence chains for stimulating as many distinct transitions as possible in the FSMs tracking the state of a block in multiple private caches. Let us illustrate that idea by means of examples. The examples assume that the program order between loads and stores to distinct locations can be relaxed, but it is certainly preserved when a memory fence is inserted between them.

Figure 5 – Examples of chain categories.



```
P1
A = 1
u = A
A = 2
x = A
```

(a) Cat. 0.

```
P1        P2
A = 1
          u = A
          A = 2
```

(b) Category 1.

```
P1        P2
A = 1
fence
u = B
          B = 2
          fence
          x = A
```

(c) Category 2.

```
P1        P2        P3
A = 1
          u = A
          fence
          B = 2
                    x = B
                    fence
                    v = A
```

(d) Category 3.

Source: adapted from Andrade, Graf & Santos (2020)

Let us first informally introduce a few notions which are required for the examples. We say that two operations collide if they access the same memory location (ADIR; SHUREK,

[2] A formal description of the algorithms can be found in (ANDRADE; GRAF; SANTOS, 2020)

2002). We say that two operations *conflict* if they collide and at least one of them is a store (GHARACHORLOO, 1995). Two operations from the same thread are in *significant program order* either if they conflict or if they are ordered by a memory fence. Two operations from distinct threads are in *significant conflict order* either if they are in conflict order or if they are colliding loads with an intervening store in conflict order with them. Different *categories* of canonical chains can be defined by such significant orderings, as illustrated in Figure 5. Upper case letters denote variables in memory, whereas lowercase letters denote variables in registers.

Figure 5a shows a uniprocessor dependence chain with all operations in significant program order. Since non-deterministic behavior is key to the exposure of shared memory errors (HANGAL et al., 2004; SHACHAM et al., 2008), Figures 5b, 5c, and 5d illustrate multiprocessor dependence chains that form data races between threads, but where operations in a same thread must execute in significant program order. Data races are formed when operations from distinct threads are in conflict order.

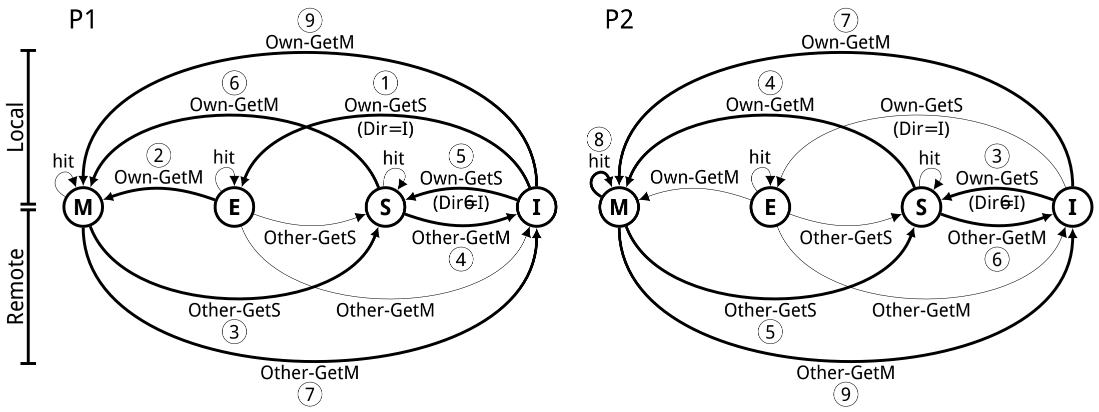
In Figure 5b, the chain constrains the first two operations and its endpoints to form data races for location A, and the last two operations to be in significant program ordering. In Figure 5c, the chain constrains the first two and the last two operations to be in significant program order (by exploiting memory fences). In Figure 5d, if the value 1 is observed for A in P2 and the value 2 is observed for B in P3, then the value 1 must be observed for A also in P3. If a multiprocessor chain is formed in execution time (as shown), the outcome of the data race involving its endpoints is deterministic, otherwise it is non-deterministic. Since each scenario induces distinct state transitions, their exploitation in different test runs tends to benefit coverage. Chains from categories 1, 2, and 3 not only drive the generator to form data races, but they also favor significant orderings. Such orderings tend to reduce the number of valid execution witnesses that do *not* lead to coherence events while the races increase the chances of detecting invalid ones. Both concur to raise the probability of error exposure and to improve coverage. That is why we exploit a mix of such categories.

The technique exploits canonical chains not for enforcing specific *consistency* rules, but for favoring proper *coherence* events instead. Figure 6 shows the conceptual connection between a canonical chain and coherence events for different protocols. Note that, as the operations in the chain are executed, an intra-processor conflict leads to local requests that induce *distinct* transitions in the local FSM, while an inter-processor conflict leads to local and remote requests that also induce *distinct* transitions. Thus, the chain's *alternation* between intra- and inter-processor conflicts tends to induce different transitions, which favors transition coverage. Since distinct protocols have similar responses for the same coherence transactions (except for a few transitions and write-back actions), this general property of a canonical chain makes the impact of the technique largely *independent* of the protocol implemented in a given design.

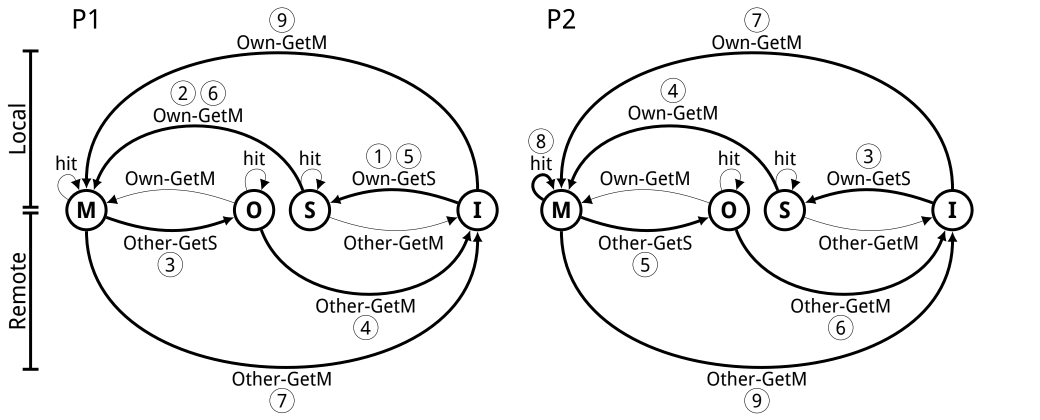
Figure 6 – How a canonical chain improves the coverage of coherence events.

	P1	P2	
①	u = A		
②	A = 1		
		u = A	③
		A = 2	④
⑤	x = A		
⑥	A = 3		
		A = 4	⑦
		x = A	⑧
⑨	A = 5		

(a) Canonical chain.



(b) MESI Protocol.



(c) MOSI Protocol.

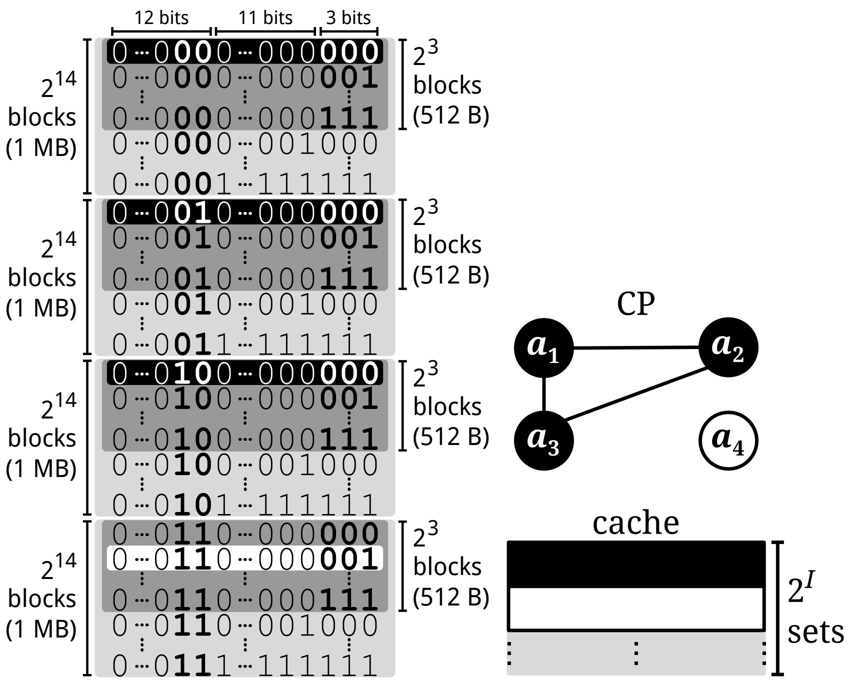
Source: adapted from Andrade, Graf & Santos (2020)

4.1.2 Address assignment: biasing

The main idea behind address assignment is the *competition biasing constraint* (*cbc*). Let us illustrate that idea by means of an example. Figure 7a shows a layout corresponding to a 32-bit address space, but where 26-bit block addresses are actually represented, because it assumes blocks with 64 bytes.

Suppose that the address range is limited to 4 MB, and it is partitioned into segments with 1MB each, as depicted by the light gray boxes. Suppose, however, that the address space to

Figure 7 – An example of address assignment.



(a) Address space constraints. (b) A CP induced by $cbc = (2,3)$.

Source: Andrade, Graf & Santos (2020)

be exploited by the generator is constrained to only 2 KB in total, and it is uniformly distributed over the partitions in *useful* sub-ranges with 512 B each, as depicted by the dark gray boxes. Note that, due to the 1 MB partitioning, the block addresses can accommodate indices for identifying up to 2^{14} distinct cache sets. In spite of that, the constraint imposed on useful sub-ranges leads to a bound of 8 for the number of different index identifiers. Besides, the constraint imposed on the amount of partitions of the full address range leads to a bound of four for the number of distinct tags identifiers. This way of restricting the address space into a few useful chunks (as depicted by the dark gray boxes) is sometimes exploited (e.g. Elver & Nagarajan (2016)) as a static address space constraint for fostering replacement events.

Instead, we propose a biasing technique that dynamically exploits address space constraints to foster replacement events *without* the need for restricting addresses to useful sub-ranges, as explained next. Consider the black and white boxes lying inside the dark gray boxes. They represent the assignment of effective addresses to four distinct shared locations. Such assignment can be seen as an instance of a general pattern such that three locations compete for the same cache set, and a single location that does not compete with the others, because it is mapped to a distinct cache set. Similar assignment instances could be induced by such same *competition pattern* (CP) within the useful address space.

For a given number of locations, a $cbc = (k, \chi)$ specifies patterns where all locations map to one out of k cache sets, and at most χ locations map to the same cache set. For instance, the address assignment illustrated in Figure 7a was induced by $cbc = (2,3)$. Figure 7b illustrates the notion of CP for this *cbc* and a scenario with four locations and a cache with 2^I sets (where I can be any positive integer). Locations mapping to the same set are represented as a clique of an undirected graph. Therefore, given a $cbc = (k, \chi)$ inducing a CP, k and χ could be seen as clique cover and chromatic numbers, respectively.

Note that *cbc* constraints can be exploited for inducing cache evictions. Given an n-way cache, a block is evicted iff $n+1$ distinct and successive addresses compete for the same set; therefore, $\chi \geq n+1$ is a necessary condition for cache eviction. Besides, k defines the number of distinct cache sets accessed by a test program.

Another desirable property of an address assignment is specified by the *sharing biasing constraint* (*sbc*). The address assignment in Figure 7a can be used to illustrate this notion. Note that the locations competing for the same cache set have distinct block addresses: despite the same index, their tags are all different. Since unrelated shared variables are not stored in the same memory block, such assignment precludes false sharing. Actually, the *sbc* is a Boolean value specifying if true sharing must be enforced or not.

Yet another desirable property is specified by the *alignment biasing constraint* (*abc*). The *abc* is a natural number specifying that all effective addresses must be aligned to 2^{abc}-byte boundaries. For instance, if we enforce the alignment to 2^6 bytes, the six offset bits implicit in Figure 7a must be zero for all effective addresses to be exploited by the generator.

The motivation for constraining the mapping of locations to effective addresses lies in the control of replacement events. For instance, the alternation between *cbc*s enabling and disabling block replacement tends to avoid revisiting the same state transition, which favors coverage and the probability of exposing design errors. This property is further explored in the Directing Engine, as explained in the next section.

4.2 DIRECTING ENGINE

The novel technique proposed in Andrade et al. (2018) is adopted in the directing engine. It exploits the constraints described in the previous chapter to enable the formulation of a simple and yet useful *coverage model*. It reveals a new mechanism to improve the quality of non-deterministic tests. The exploitation of general properties of coherence protocols and cache memories allows better control on transition coverage, which serves as a proxy for increasing the actual coverage metric adopted in a given verification environment. Being independent of coverage metric, coherence protocol, and cache parameters, the directing engine is reusable across quite different designs and verification environments. The main features of the adopted technique are: (1) the casting of *general* properties of coherence protocols and cache memories as non-conventional constraints on RTG; (2) a new coverage model resulting from such constraints, which serves as a proxy for whatever coverage metric is adopted; and (3) a novel, *steep coverage-ascent* (SCA) algorithm for the directing engine, which relies on the proposed coverage model, and whose behavior will be illustrated by means of an example.

4.2.1 Coverage control: steep coverage ascent

The approach relies on a *classification of state transitions* to specify the coverage model. Given the FSM that specifies the behavior of a coherence protocol for a given cache controller, three classes of transitions are distinguished:

- *Class 1*: transitions induced by local events triggered by the core or by another private cache controller lying on the same core domain at the immediate higher hierarchical level. Such events result from intra-processor collisions.
- *Class 2*: transitions induced by local events triggered by requests from remote cores. Such events result from inter-processor collisions.
- *Class 3*: transitions induced by replacement events triggered by the controller itself.

Besides, it exploits constraints on RTG as mechanisms for enabling better control on coverage improvement. The first constraint enforces the alternation between Class 1 and Class 2 transitions, which is exploited by the RTG engine itself. The second one paves the way to the alternation between Class 3 and Classes 1/2 transitions, which is exploited by the directing engine, as follows.

Constraint 1: enforcement of *alternation* between Classes 1 and 2. To increase the chances of raising transition coverage, the RTG engine builds each test program according to rules that make successive colliding accesses likelier to induce transitions that are *different* from those already covered, as illustrated in Section 4.1.1.

Constraint 2: enforcement of uniform competition. Given the $cbc = (k, \chi)$ defined in Section 4.1.2, we assign $\chi = s/k$ for each value of s. Then, we limit the values that can be assigned to k: only those for which s is multiple of k are kept in the generation space. As a result, the RTG engine is constrained to assign effective addresses in such a way that exactly s/k locations compete for each cache set. Such *uniform* distribution maximizes the probability of inducing replacements in all sets for a given setting of a pair (s, k).

Given the Constraint 2, the parameter k can be used to enabling or disabling Class 3 transitions, as follows. Let α denote the associativity of a cache. For inducing a replacement event in a given cache set, a sequence of at least $\alpha + 1$ references to *distinct* locations competing for that set is required. Therefore, a necessary condition for enabling replacement is $s/k \geq \alpha + 1$. Conversely, a sufficient condition for disabling replacement in all sets is $s/k < \alpha + 1 \Leftrightarrow s/k \leq \alpha$. Thus, there is a threshold s/α for the value of k above which replacement is certainly disabled, but below which it may be enabled depending on the sequence of references that turns out to be generated randomly. Therefore, the RTG engine can stimulate the alternation between Class 3 and Class 1/2 transitions by selecting appropriate values of k.

The exploitation of Constraints 1 and 2 results in a useful coverage model, which estimates the coverage of each class based on the values of n, s and k. We pessimistically assume that the Classes 1, 2, and 3 induce a partition of the set of transitions of a FSM. Let

$TC = (tran_1 + tran_2 + tran_3)/total = TC_1 + TC_2 + TC_3$ denote the transition coverage of the FSM specifying the protocol behavior for the memory block containing a given location, where $tran_j$ denotes the number of distinct transitions from Class j and $total$ denotes the overall number of transitions in the FSM. Besides, every collision leads to a transition from Classes 1 or 2, i.e., $TC_{1/2} = TC_1 + TC_2$ denotes the coverage of collision-induced transitions. With such definitions, the coverage model can be expressed with the following relations[3]:

$$\widehat{TC}_{1/2} \propto n/s$$

$$\widehat{TC}_3 \propto \begin{cases} (n/k)/(\alpha+1) \text{ (upper bound)} \\ (s/k)/(\alpha+1) \text{ (lower bound)} \end{cases}$$

Based on this model, the directing engine iteratively selects parameters (n,s,k) to command the RTG engine, as exemplified in the next section.

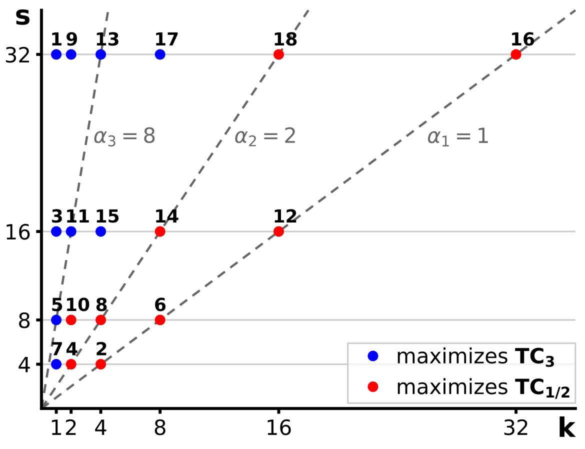
4.2.2 An example of how it works

The values assigned to the parameters (n,s,k) induce a tridimensional generation space. The directing engine explores planes of such space, which are induced by assigning increasing values to the parameter n.

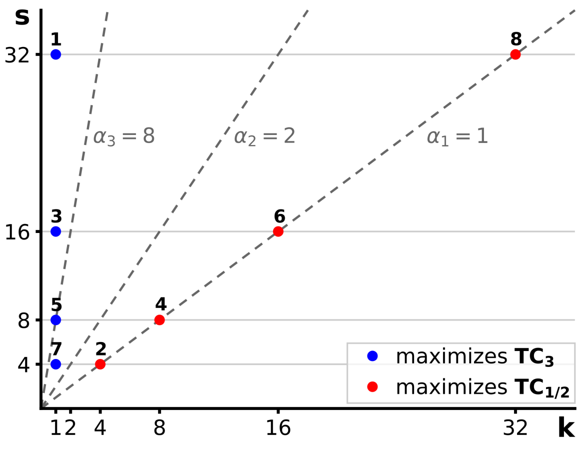
Figure 8a illustrates one such plane for a range of shared locations defined as $S = \{4,8,16,32\}$. Let us suppose a memory hierarchy with three cache levels, each with a distinct associativity: L1 is directed mapped, L2 is 2-way, and L3 is 8-way. A point with a mark (square, triangle or circle) represents a pair (s,k) that leads to a uniform distribution of location competition for cache sets. Unmarked points were excluded from the generation space by Constraint 2. Dashed lines correspond to the distinct degrees of associativity at each cache level. Each dashed line groups the points of the generation space that represent the threshold for disabling replacement events for different values of s. For a given number of locations s, a mark in a dashed line labeled as α_j represents the minimum value of k required for disabling replacement events in a α_j-way cache lying at level j. Therefore, the marks to the left of a dashed line denote the values of k that are likely to stimulate replacement-induced transitions at level j (Class 3), while all the marks to the right (or on the line itself) denote values of k that certainly do not induce replacement events in any set of a cache at level j, being therefore likely to stimulate collision-induced transitions instead (Class 1/2). For this reason, when trying to stimulate collision-induced transitions, the directing engine could explore only the pairs marked with circles, because the value $k = s$ is sufficiently large for disabling replacement-induced transitions at *all* levels for a given s. On the other hand, when trying to stimulate replacement-induced transitions, the engine could explore only the pairs marked with squares, because the value $k = 1$ corresponds to the maximum probability of replacement for a given s. Note, however, that albeit the pairs marked with

[3] The steps leading to such formulation are detailed in (ANDRADE et al., 2018).

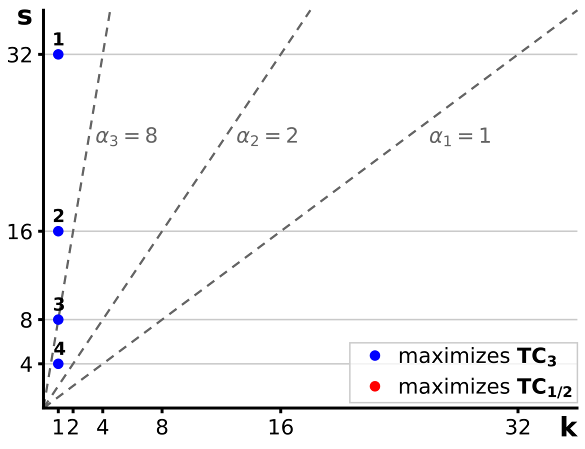
Figure 8 – How a plane of the generation space is searched.



(a) Search space for Variant 1.



(b) Search space for Variant 2.



(c) Search space for Variant 3.

Source: adapted from Andrade et al. (2018)

squares are likely to enable replacement at most levels, this may not necessarily hold for all (for instance, $(4,1)$ and $(8,1)$ may enable replacement at L1 and L2, but not at L3).

Therefore, to control the stimulation of a desired type of transition, the directing engine does not necessarily have to explore all the marked points in Figure 8a, but only search the subspace marked with squares and circles. Figure 8b illustrates such a reduced search space. Note that, any traversal alternating between square and circle is likely to stimulate a sequence of transitions induced by an alternation between replacement and collision events. To select the most convenient traversal, the directing engine relies on the proposed coverage model. Since $\widehat{TC}_3 \propto s/k$ or $\widehat{TC}_3 \propto n/k$, when exploring a plane for a given n, the engine selects the pair in the search space with minimum k and maximum s. Since $\widehat{TC}_{1/2} \propto n/s$, the engine selects the pair in the search space with minimum s. Albeit in such case k could be arbitrary selected according to the coverage model, the engine chooses the maximal k, because it has the advantage of disabling replacement at all levels. Such choices lead to the traversal indicated by the labeling in Figure 8b. Note that, in such a traversal, a move from a square to a circle, corresponds to the alternation between the maximum probability of replacement and the maximum probability of collision for a given *unexplored sub*-space. Thus, such *steep coverage-ascent* traversal was designed to reach the highest coverage as possible in the smallest time (thereby reducing the effort to find errors), while still exploring the tridimensional search space by successively visiting planes induced by increasing values of *n* (for reaching the highest coverage as possible within a pre-specified range). The labeling in Figure 8a illustrates such a traversal for the original search space. Finally, Figure 8c illustrates a degeneration of it for an even smaller search space, which fosters Class 3 transitions predominantly. Any of the three variants illustrated in Figure 8 can be used by the directing engine[4].

Since the role of a checker in the framework and the potential impact of proper generation techniques to MCM checking are now clear, the next chapter can focus on the details of the proposed approach to building checkers.

[4] A formal description of the algorithm underlying the directing engine can be found in Andrade et al. (2018).

5 MCM-BASED RUNTIME CHECKING

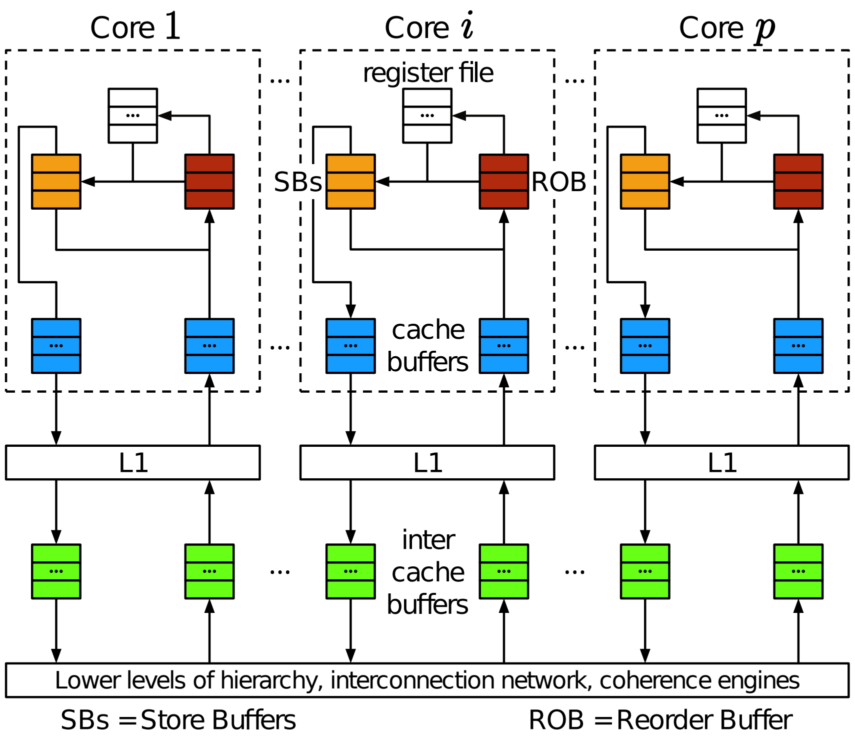
This chapter proposes a novel approach to the building of runtime checkers. The approach allows the tailoring to distinct architectures and to different microarchitecture variants, and it relies on two key ideas:

- An *abstract specification* for building checkers targeting relaxed memory models that comply with either rMCA or nMCA stores. It partitions the memory behavior into architecture-dependent and architecture-independent axioms (the former being reusable; the latter, deserving target customization).

- An *observability template* that pinpoints where to insert monitors in the design representation such that the observed physical events can be used as proxies for the abstract events from the axiomatic specification. It is largely independent of microarchitecture, because it restricts monitors to structures common to most implementations: L1 cache buffers and reorder buffers (or similar structures for handling out-of-order execution). As a result, the dependence on design is narrowed down to monitor implementation, and it does not affect monitor location.

The approach was designed to be general enough for checking multicore chips based on superscalar microarchitectures with out-of-order execution and hardware-based speculation,

Figure 9 – Generic model to accommodate the target architectures.



Source: adapted from Graf et al. (2019)

such as IBM Power, and aggressive implementations of ARMv8 and RISC-V architectures. To accommodate such microarchitectures, we adopt a generic model, as illustrated in Figure 9[1].

The model consists of a design with *p* cores, each accessing a private cache. Only the relevant flow of data is explicitly shown (the flow of addresses is omitted for simplicity). We assume the following types of buffers: (1) a *reorder buffer* (ROB) (HENNESSY; PATTERSON, 2017) (or similar structure for instruction commit in program order), (2) *store buffers* (or similar structures for keeping addresses and values of outstanding stores), (3) incoming and outgoing *cache buffers* (for keeping local memory requests and replies), (4) incoming and outgoing *inter-cache buffers* (for keeping coherence messages to or from other cores, either invalidate (update) requests or data replies). We abstract lower hierarchical levels, interconnection network, and coherence engines.

The rest of this chapter is organized as follows. Section 5.1 introduces the notion of abstract events and formalizes the axioms of the abstract specification. Section 5.2 shows how the observability template maps abstract events to physical events, pinpointing the adequate places to monitor such events. Finally, Section 5.3 shows how a few properties and constraints can be exploited for building efficient checkers.

5.1 ABSTRACT SPECIFICATION

This section proposes a specification that captures both pipeline and shared memory effects (e.g. out-of-order execution, coherence, and consistency) on the behavior of load and store operations. It consists of a set of axioms that comply with modern architectures relying on relaxed ordering and nMCA stores. To improve comprehension, pictorial representations are used to clarify the formal axioms. Section 5.1.1 defines the *abstract* memory events used in the specification. Section 5.1.2 proposes axioms capturing the aspects of memory behavior that are common to most modern architectures (e.g. write serialization and relaxed ordering). Section 5.1.3 shows how to extend the specification for capturing the aspects of memory behavior that are specific to some architecture (e.g. fences).

5.1.1 Abstract events

We adopt the following granularity for the memory events: (1) each store is split into multiple copies, one for each core, (2) each copy is further split into commit and completion phases. To capture pipeline effects on memory behavior, (3) each load is single-copy, and (4) each load is split into commit and completion phases. Gharachorloo (GHARACHORLOO,

[1] To be general, the model has to assume complex cores requiring reorder buffers (or similar structures). However, if the design under verification happens to be a multicore chip relying on simpler cores (e.g. in-order pipelines), the model degenerates into a simpler structure. The simpler target design does not limit the proposed approach (it just simplifies the axioms of the abstract representation and the observability template). Indeed, MCM checking is mainly about shared *memory* behavior, not about core behavior. Our capturing of inner core behavior is just a means to overcome side effects of dynamic scheduling and speculation when they would hamper shared-memory verification.

1995) relied on (1) for proposing a formal rMCA-compliant specification, and on (2) and (3) for *informally* picturing how nMCA stores could be handled, but not on (4), since loads were considered atomic. In contrast to (GHARACHORLOO, 1995), we directly combine all the fine-grain events resulting from (1), (2), (3) and (4) into a single *formal* abstract specification that is suitable for functional verification.

Consider an *operation* O_j, issued by core i, which makes reference to some location a, written $(O_j)^i_a$. Let L or S replace O to denote that the operation is either a load or a store, respectively. An operation gives rise to multiple *events*, but not all of them are relevant for specifying memory behavior. That is why *abstract* events are usually employed for specification (ADVE; HILL, 1992; GHARACHORLOO, 1995). They represent the time when stores and loads take effect with respect to a given core, i.e., they represent the ultimate effect of a chain of *physical* events throughout the memory hierarchy. We adopt the following abstract events.

A load operation issued by a core i, say $(L_j)^i_a$, gives rise to at most two relevant events: a *read completion* event $(R_j)^i_a$ and, possibly, a *read commit* event $(r_j)^i_a$. The former represents the reading of a value either from cache or from a local buffer (read forwarding); the latter, its storage into the register file.

Let p be the number of cores in a multicore chip. A store operation issued by a core i, say $(S_j)^i_a$, gives rise to p *write commit* events $(w_j)^x_a$ and p *write completion* events $(W_j)^x_a$, with $x = 1, \cdots, p$. Note that $(w_j)^i_a$ and $(W_j)^i_a$ are commit and completion events with respect to the core that actually issued S_j. The former represents the placement of a (non-speculative) value into an outgoing buffer (as part of an outstanding write request); the latter, the actual writing into cache. Note that $(w_j)^{x\neq i}_a$ and $(W_j)^{x\neq i}_a$ are events induced by cache coherence. The former represents the placement of an invalidate (update) request into an incoming buffer of core x; the latter, the actual cache block invalidation (update).

An abstract event $(W_j)^{x\neq i}_a$ may represent distinct physical events. For instance, if core x has a copy of the block at L1, $(W_j)^{x\neq i}_a$ corresponds to a physical event at the L1 cache's interface. Otherwise, $(W_j)^{x\neq i}_a$ corresponds to a physical event that guarantees the completion with respect to core x at a lower level, e.g. $(W_j)^{x\neq i}_a$ may represent invalidation (update) at the L2 cache.

To capture the *availability* of a value from a store $(S_j)^i_a$ to a subsequent conflicting load, before it is written to memory, we let $(\omega_j)^i_a$ denote an event representing the placement of a value into a store buffer, before $(S_j)^i_a$ commits in core i.

To describe memory behavior in terms of abstract events, we rely on the following notation. Given two (load or store) instructions I_j and I_m, if I_j precedes I_m in some thread, we say that their respective operations are in program order, written $O_j \prec_{po} O_m$. $\mathscr{O}$ is the set of memory operations issued by all p cores, $\mathscr{S}$ is the set of all stores, and $\mathscr{L}$ is the set of all loads. $\mathscr{O}^i \subset \mathscr{O}$ ($\mathscr{S}^i \subset \mathscr{S}$, $\mathscr{L}^i \subset \mathscr{L}$) are operations induced by the instructions issued by some core i. $\mathscr{O}^i_a \subset \mathscr{O}^i$ ($\mathscr{S}^i_a \subset \mathscr{S}^i$, $\mathscr{L}^i_a \subset \mathscr{L}^i$) are subsets of operations colliding at the same location a. We drop a subscript or superscript when the location or the issuing core is irrelevant. We let Val^0_a be the initial value stored at location a before any core ever writes to it.

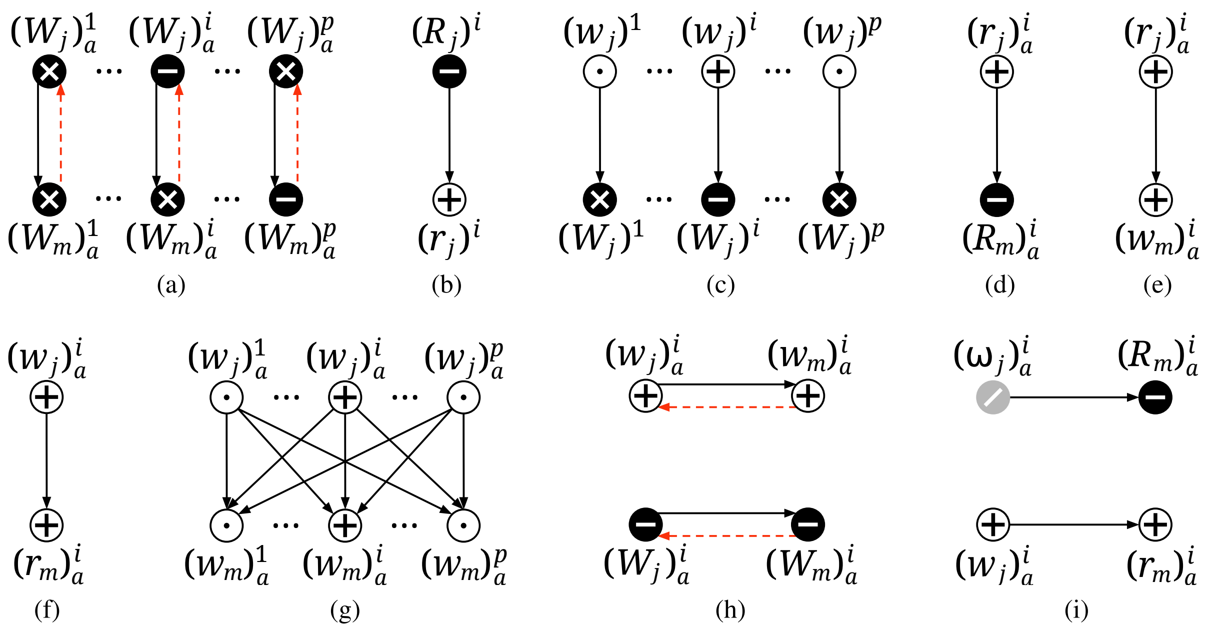
An *execution witness* represents a distinct outcome of a parallel program, and it is determined by which store operation is observed by each load. An execution induces some memory *behavior*, which can be seen as a totally ordered set of memory *events* (GHARACHORLOO, 1995). Every valid behavior must satisfy a partial order $\leq$ on the set of memory events. Valid memory behaviors are defined by ordering constraints, which are imposed by cache coherence, data dependencies, and to which extent program order is preserved by memory consistency.

It is well known that non-deterministic parallel programs tend to expose shared memory bugs faster than real-life, synchronized programs (MANOVIT; HANGAL, 2006; ELVER; NAGARAJAN, 2016). Therefore, test generation is often constrained to synthesize simple, non-deterministic programs stressing memory accesses (as shown in Chapter 4). For simplicity, let us assume that the instruction sequences of a synthetic program: (1) do not contain locks, (2) may form data dependencies through memory, but not through registers (i.e., neither address nor source register dependencies are synthesized), and (3) do not form control dependencies (but are still suitable to speculation on address disambiguation). The next two sections formalize axioms that specify valid behaviors under such assumptions.

5.1.2 Architecture-independent behavior

To illustrate the axioms, Figure 10 pictures commit, completion, and availability events as white, black, and gray circles, respectively. Besides, the symbols $\oplus$ and $\ominus$ denote events at the issuing core, whereas $\otimes$ and $\odot$ denote events induced at remote cores.

Figure 10 – Pictorial representations for Axioms 1 to 6.



$(W_j)_a^1$ $(W_j)_a^i$ $(W_j)_a^p$
$(W_m)_a^1$ $(W_m)_a^i$ $(W_m)_a^p$
(a)
$(R_j)^i$
$(r_j)^i$
(b)
$(w_j)^1$ $(w_j)^i$ $(w_j)^p$
$(W_j)^1$ $(W_j)^i$ $(W_j)^p$
(c)
$(r_j)_a^i$
$(R_m)_a^i$
(d)
$(r_j)_a^i$
$(w_m)_a^i$
(e)
$(w_j)_a^i$
$(r_m)_a^i$
(f)
$(w_j)_a^1$ $(w_j)_a^i$ $(w_j)_a^p$
$(w_m)_a^1$ $(w_m)_a^i$ $(w_m)_a^p$
(g)
$(w_j)_a^i$ $(w_m)_a^i$
$(W_j)_a^i$ $(W_m)_a^i$
(h)
$(\omega_j)_a^i$ $(R_m)_a^i$
$(w_j)_a^i$ $(r_m)_a^i$
(i)

Source: Graf et al. (2019)

Coherent systems require that all write events to the same location complete in exactly the same order from the perspective of every core, albeit the copies of a given store may *not* be observed as atomic (i.e., nMCA) in each core, as follows.

Axiom 1. ***Cache coherence serialization constraint***[2] *(Figure 10a):*
$\forall S_j, S_m \in \mathscr{S}_a : (\forall_{x=1}^{p} : (W_j)_a^x \leq (W_m)_a^x) \vee (\forall_{x=1}^{p} : (W_m)_a^x \leq (W_j)_a^x)$

It should be noted that Axiom 1 *minimally* constrains the serialization of write events, because it assumes nMCA store behavior.

Pipeline effects on load and store operations are captured as follows. A load completes before it may be committed, a store commits before it completes with respect to every core, i.e.:

Axiom 2. ***Load completion-commit constraint*** *(Figure 10b):*
$\forall L_j \in \mathscr{L} : R_j \leq r_j$

Axiom 3. ***Store commit-completion constraint*** *(Figure 10c):*
$\forall S_j \in \mathscr{S} : \forall_{x=1}^{p} : (w_j)^x \leq (W_j)^x$

The order between operations to distinct locations has been fully relaxed in modern architectures, but program-order *commit* must be preserved at least for operations colliding at a same location. *Completion* order between *conflicting*[3] operations (GHARACHORLOO, 1995) can be relaxed as long as WAR, WAW, and RAW hazards[4] (HENNESSY; PATTERSON, 2017) are prevented in the scope of each thread (whether through memory or through read forwarding). Besides, program order must be preserved for loads colliding at a same location, because illegal outcomes would otherwise occur in lock-free programs if stores conflicting to the same location were observed by those loads (ALGLAVE; MARANGET; TAUTSCHNIG, 2014; ARM, 2011). For proper ordering of colliding loads, the commit of the first one must occur before the completion of the second[5]. This is formalized below.

Axiom 4. ***Preserved program order constraints*** *(Figure 10d-g):*
For all $O_j, O_m \in \mathscr{O}_a^i$, the following must hold in the case of

- $L_j \prec_{po} L_m : (r_j)_a^i \leq (R_m)_a^i$
- $L_j \prec_{po} S_m : (r_j)_a^i \leq (w_m)_a^i$
- $S_j \prec_{po} L_m : (w_j)_a^i \leq (r_m)_a^i$
- $S_j \prec_{po} S_m : \forall x,y : (w_j)_a^x \leq (w_m)_a^y$

[2] In contrast, for an rMCA target, there must be a *global* linear order of conflicting (atomic) stores, i.e., $\forall_{x,y} : ((W_j)_a^x \leq (W_m)_a^y) \vee ((W_m)_a^x \leq (W_j)_a^y)$.

[3] Recall that two operations conflict if they collide at the same location and at least one is a store (GHARACHORLOO et al., 1990).

[4] As specified (later) by Axioms 4 (clause 3), 5, and 6, respectively.

[5] As $(r_j)_a^i \leq (R_m)_a^i \wedge (R_m)_a^i \leq (r_m)_a^i \Rightarrow (r_j)_a^i \leq (r_m)_a^i$, Axiom 4, clause 1, combined with Axiom 2 results in a necessary and sufficient condition for preventing the so-called "RAR hazard" acknowledged as a bug in (ARM, 2011).

The last clause of Axiom 4 ensures that conflicting stores are committed in program order by waiting for the first store to commit with respect to every core before the second is committed with respect to any core (GHARACHORLOO, 1995). Note that the combination of the second clause of Axiom 4 with Axioms 2 and 3, rules out WAR hazards through memory[6]. To prevent WAW hazards through memory, the actual updating of memory must occur in program order (HENNESSY; PATTERSON, 2017), i.e.:

Axiom 5. ***Local store completion order constraint*** *(Figure 10h):*
$\forall S_j, S_m \in \mathscr{S}_a^i : (w_j)_a^i \leq (w_m)_a^i \Rightarrow (W_j)_a^i \leq (W_m)_a^i$

RAW hazards must be prevented through memory *and* read forwarding. Thus, for every conflicting store preceding a load in program order, the value to be written by the store must have been made available in a local buffer, before the actual reading takes place[7]:

Axiom 6. ***Store value availability constraint:*** *(Figure 10i):*
$\forall S_j, L_m \in \mathscr{O}_a^i : (w_j)_a^i \leq (r_m)_a^i \Rightarrow (\omega_j)_a^i \leq (R_m)_a^i$

Now, let us define how a load L_a^i observes the value it returns. We first formalize a few notions for paving the way towards a new axiom, whose pictorial representation is shown in Figure 11. L_a^i may observe the value made available by some conflicting store that precedes the load in program order but has not completed with respect to core *i* before the reading takes place, i.e.:

Definition 1. The set of stores locally observable by L_a^i is
$\sigma_L(L_a^i) = \{S_j \in \mathscr{S}_a^i : (w_j)_a^i \leq r_a^i \wedge (\omega_j)_a^i \leq R_a^i \leq (W_j)_a^i\}$.

The second clause of Definition 1 specifies that a store's value must be *available* before load *completion* (to enable read forwarding), whereas the first clause requires that the store commits before the load (to preserve program order). Thus, since a load may only observe a value made available from a store preceding it in program order, Definition 1 rules out WAR hazards through read forwarding.

L_a^i may observe the value from some store that completed with respect to core *i* (whether it was locally issued by *i* itself or by another core $x \neq i$), i.e.:

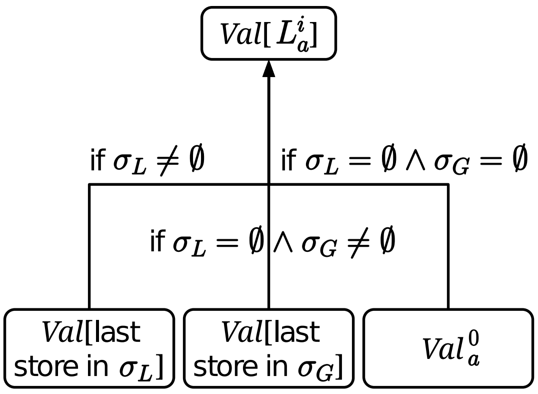
Definition 2. The set of stores globally observable by L_a^i is $\sigma_G(L_a^i) = \{S_j \in \mathscr{S}_a : (W_j)_a^i \leq R_a^i\}$.

Definition 3. The last store locally observed by L_a^i, denoted as $Max[\sigma_L(L_a^i)]$, is the operation $S_j \in \sigma_L(L_a^i)$ such that $\forall S_x \in \sigma_L(L_a^i) : (w_x)_a^i \leq (w_j)_a^i \leq r_a^i$.

Definition 4. The last store globally observed by L_a^i, written $Max[\sigma_G(L_a^i)]$, is the operation $S_j \in \sigma_G(L_a^i)$ such that $\forall S_x \in \sigma_G(L_a^i) : (W_x)_a^i \leq (W_j)_a^i \leq R_a^i$.

[6] $(R_j)_a^i \leq (r_j)_a^i \wedge (r_j)_a^i \leq (w_m)_a^i \wedge (w_m)_a^i \leq (W_m)_a^i \Rightarrow (R_j)_a^i \leq (W_m)_a^i$.
[7] This is a precondition for proper value consumption, whether it is obtained from memory or through read forwarding. The appropriate choice is specified by Axiom 7.

Figure 11 – Pictorial representation of Axiom 7.


$Val[L_a^i]$
if $\sigma_L \neq \emptyset$
if $\sigma_L = \emptyset \wedge \sigma_G = \emptyset$
if $\sigma_L = \emptyset \wedge \sigma_G \neq \emptyset$
Val[last store in σ_L]
Val[last store in σ_G]
Val_a^0

Source: the author.

Let $Val[(O_j)_a^i]$ be the value written or returned by some operation issued by core i. The value returned by L_a^i corresponds to the last[8] observed store, unless none was observed, when the initial value is returned, as follows.

Axiom 7. ***Returned value constraint*** *(Figure 11):*
$\forall L \in \mathscr{L}_a^i$:

$$Val[L_a^i] = \begin{cases} Val[Max[\sigma_L(L_a^i)]] & if\, \sigma_L(L_a^i) \neq \emptyset \\ Val[Max[\sigma_G(L_a^i)]] & if\, \sigma_L(L_a^i) = \emptyset \wedge \sigma_G(L_a^i) \neq \emptyset \\ Val_a^0 & if\, \sigma_L(L_a^i) = \emptyset \wedge \sigma_G(L_a^i) = \emptyset \end{cases}$$

5.1.3 Architecture-dependent behavior

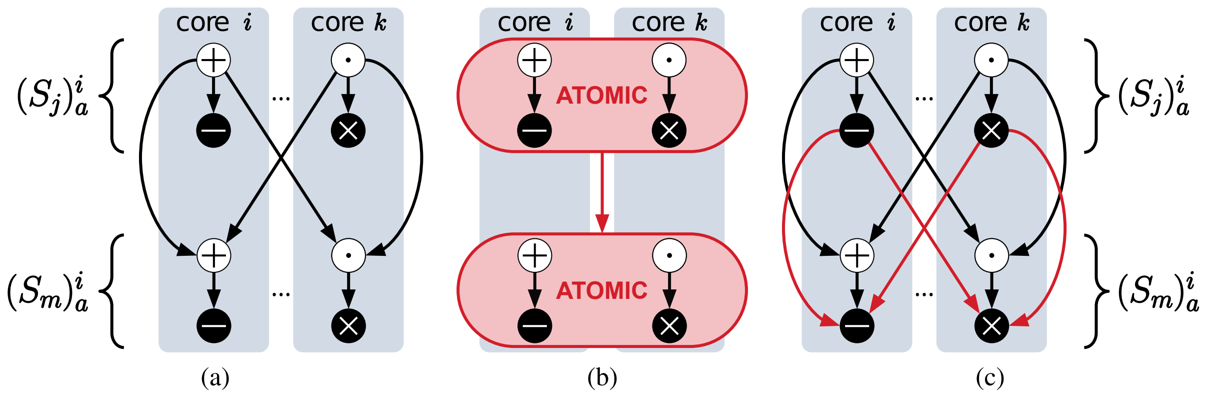
It should be noted that Axioms 1 to 7 represent *minimal* constraints on shared memory behavior, that is, they must be satisfied by all architectures. If an architecture (or microarchitecture) happens to enforce stronger constraints, new axioms will be required to capture them. This section provides two examples of extra ordering constraints. The first one shows how to restore program order for architectures relying on relaxed MCMs. The second one illustrates how to enforce rMCA store behavior when required.

Fences are sometimes used to restore program order between operations that are not data-dependent (non-cumulative behavior), and sometimes they also guarantee proper ordering in chains of memory operations spanning multiple threads (cumulative behavior). This dissertation focuses on non-cumulative behavior[9], because the complexity of supporting cumulative fences does not pay off for most multicore chip designs (PULTE et al., 2017; WATERMAN; ASANOVI, 2019).

[8] The last store is determined either by program order (Axiom 4) or by store serialization (Axiom 1), which both induce total orders of conflicting stores.

[9] However, our abstract specification is able to encode such complementary behavior in an extra axiom, which is omitted in this dissertation, but can be found in (GRAF et al., 2019).

Figure 12 – How nMCA and rMCA behaviors are captured by the axioms.



Source: the author.

Axiom 8 captures the non-cumulative behavior of a fence F, which is similar to Power's `hwsync` and to ARM's `dmb isb`[10].

Axiom 8. ***Program order constraint on operation completion:***
For all $O_j, O_m \in \mathscr{O}^i$, the following must hold in the case of

- $L_j \prec_{po} F \prec_{po} L_m : (R_j)^i \leq (R_m)^i$
- $L_j \prec_{po} F \prec_{po} S_m : (R_j)^i \leq (W_m)^i$
- $S_j \prec_{po} F \prec_{po} L_m : (W_j)^i \leq (R_m)^i$
- $S_j \prec_{po} F \prec_{po} S_m : (W_j)^i \leq (W_m)^i$

Let us now show how to enforce rMCA behavior. As mentioned before, high performance microarchitectures may allow nMCA store behavior under the covers through aggressive optimizations, as long as it is not exposed to the programmer (WATERMAN; ASANOVI, 2019). Since a runtime MCM checker requires multiple monitors to provide verification guarantees without compromising scalability with growing core counts, it ends up being exposed to such hidden nMCA behaviors. That is why our approach to the building of checkers provide the required flexibility to model both, rMCA and nMCA store behaviors, as illustrated in Figure 12.

In designs with relaxed store atomicity (nMCA store behavior), if two conflicting stores are issued in program order by the same core, the first store must commit with respect to every core, before the second one is committed with respect to any core, as formalized by Axiom 4, Clause 4, and depicted in Figure 12a. In other words, the second store can only be committed after the first one has placed a write request into the outgoing buffer of the local core and invalidate requests into the incoming buffers of remote cores. This means that, the second store does not have to wait for invalidations to take place, but only for early acknowledgements from every remote core incoming buffer (GHARACHORLOO, 1995).

[10] For fences similar to Power's `lwsync` and ARM's `dmb ishld` and `dmb ishst`, we must remove, respectively: clause 3 only, both clauses 3 and 4, and all clauses but 4.

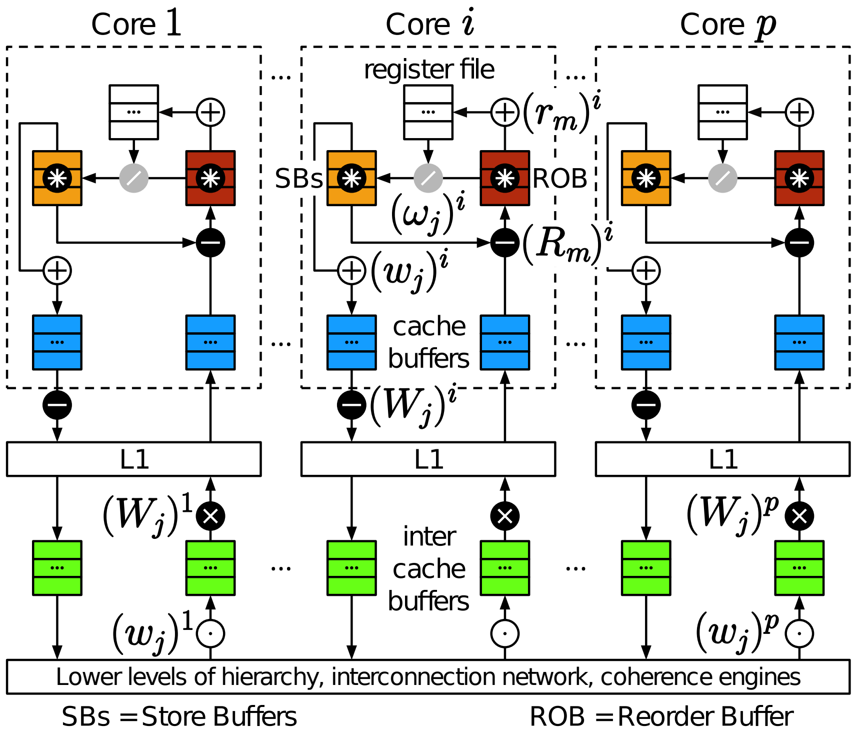
In contrast, in designs with strict store atomicity (rMCA store behavior), the commit and completion events of the first store must appear as ordered before the commit and completion events of the second store, creating the illusion of being atomic, as shown in Figure 12b. This can be achieved by an additional ordering constraint on completion events. Since our representation deliberately decouples coherence from issue requirements, and it specifies *minimal* constraints, stronger conditions can be captured by additional constraints, as illustrated in Figure 12c, and formalized below.

Axiom 9. ***Preserved store atomicity constraint*** *(Figure 12c):*
$\forall S_j, S_m \in \mathscr{S}^i : S_j \prec_{po} S_m \Rightarrow \forall x, y : (W_j)_a^x \leq (W_m)_a^y$

5.2 OBSERVABILITY TEMPLATE

This section proposes a template that can accommodate most microarchitectures, because physical events are monitored only at the interface with buffers common to most dynamically scheduled pipelines and with buffers at the *first* cache level only[11]. It defines *which* physical events should be used as *proxies* for abstract events, and *where* they should be observed for proper axiom verification. Figure 13 shows the proposed template for a design with p cores, based on the model depicted in Figure 9.

Figure 13 – Observability template for a generic multicore chip design.



Source: adapted from Graf et al. (2019)

[11] This makes checkers independent from the number of levels in a design.

Every *monitor* captures a physical event as a triple (op, a, v), where op is either write or read, a is a location, and v is a value (if any). Thus, the distinction between commit and completion events and between local and remote events can only be made by inserting monitors at adequate points. That is why we use distinct monitor types in Figure 13, denoted as $\oplus$, $\ominus$, $\odot$, $\otimes$, $\oslash$, and $\circledast$. They indicate the relevant points for observing memory events in each core domain. Let us show how a physical event observed by a monitor can be used as a proxy for a given abstract event.

Given a load L_m issued by core i, its *completion*, i.e., $(R_m)^i$, can be asserted when monitor $\ominus$ observes either a cache reply or a store buffer reply (read own write early) to a read request. Its *commit*, i.e., $(r_m)^i$, can be asserted when monitor $\oplus$ observes that L_m has reached the head of the ROB.

Given a store S_j issued by core i, its *commit* with respect to that core, i.e., $(w_j)^i$, can be asserted when monitor $\oplus$ observes the store buffer corresponding to S_j when the latter has reached the head of the ROB. Its *completion* with respect to that core, i.e., $(W_j)^i$, can be asserted when monitor $\ominus$ observes that a write request on entry to core i's cache took effect. On the other hand, the completion of S_j with respect to another core x, i. e. $(W_j)^{x \neq i}$, can be asserted when monitor $\otimes$ observes an invalidate (update) request on entry to core x's cache *if* it holds a copy of the block. If it does *not*, the completion can be asserted only (after a miss) when the owner responds to (a getM or getS) request for that block. In this case, completion can be asserted when monitor $\otimes$ observes a data reply event on entry to core x's cache. Thus, monitor $\otimes$ should take either the former (if any) or the latter (otherwise) as proxy for a write completion event. Similarly, the commit of S_j with respect to another core x, i.e., $(w_j)^{x \neq i}$, can be asserted when monitor $\odot$ observes that either an invalidate (update) request or a data reply (to getM or getS) was put into core x's incoming buffer. Finally, the *availability* of S_j's value for consumption, i.e., $(\omega_j)^i$, can only be asserted when monitor $\oslash$ observes that a value is written to the store buffer that was allocated to S_j.

The axioms specify valid behavior for operations whose instructions commit. However, when a memory instruction is discarded before reaching the head of the ROB, its record in the ROB or in a store buffer is 'squashed'. It would be impossible to tell such correct behavior from an anomaly *at runtime* if we only observed commit and completion events (with $\oplus$ and $\ominus$). That is why an extra monitor $\circledast$ observes *squash events* either at the ROB or at some store buffer for avoiding false positives due to speculation, as explained next.

5.3 BUILDING RUNTIME CHECKERS

We propose the building of checkers that monitor the physical events specified in Section 5.2, use them as proxies for abstract events, and verify whether or not their runtime behavior complies with the specified axioms. For a given architecture, it is possible to build different flavors of checkers depending on the target design, either by fully relaxing store atomicity (for nMCA), as shown in Section 5.1, or by imposing extra constraints (for MCA or rMCA) with additional axioms. Besides, even for a given target design (say nMCA), it is possible to build distinct checkers. For instance, a checker can be tailored to a subset of the specified orderings, when its complement was previously checked in prior verification steps and can thus be assumed as a *validated design property*. This section describes an example of how a few properties and constraints can be pragmatically adopted for deriving efficient checkers targeting designs with either rMCA or nMCA behaviors for a given architecture.

A checker dynamically verifies if every completion event matches either a commit event or a squash event. For all completion events matching commit events, a checker verifies if their orderings comply with the axioms, as follows. Let $\mathscr{E}$ be the set of all *physical* events monitored for a given execution witness. Let $<$ be the observed order on $\mathscr{E}$. A checker's goal is to verify if the order $<$ complies with the specified order of abstract events $\leq$. Let $\mathscr{E}^i$ be the set of *physical* events observed in the domain of core i and let us distinguish its relevant subsets: (1) $\mathscr{R}^i_\ominus$, $\mathscr{R}^i_\oplus$, and $\mathscr{R}^i_\circledast$ are the sets of read (completion, commit and squash) events; (2) $\mathscr{W}^i_\oslash$, $\mathscr{W}^i_\oplus$, $\mathscr{W}^i_\ominus$ and $\mathscr{W}^i_\circledast$ are the sets of write (availability, commit, completion, and squash) events corresponding to stores issued by core i; $\mathscr{W}^i_\odot$ and $\mathscr{W}^i_\otimes$ are the sets of write (commit and completion) events corresponding to stores issued by cores other than i.

As memory operations can be executed speculatively and their results may be discarded, the sets of outstanding loads and stores have each two relevant subsets: $\mathscr{R}^i_\ominus = R^i_\ominus \cup \overline{R}^i_\ominus$, where $R^i_\ominus$ and $\overline{R}^i_\ominus$ are the subsets of completion events corresponding to committed and squashed loads, respectively; $\mathscr{W}^i_\oslash = W^i_\oslash \cup \overline{W}^i_\oslash$, where $W^i_\oslash$ and $\overline{W}^i_\oslash$ are the subsets of availability events corresponding to committed and squashed stores, respectively.

To face the huge number of valid execution witnesses resulting from a largely relaxed memory model, while preserving verification quality, let us assume that a few properties are known to hold before shared memory verification is launched (because they were validated during processor design). The rationale is that, by checking for errors that are easier to find in advance, the dynamic checker can focus on efficiently uncovering more subtle errors.

Let us assume that issue units and commit units work properly, i.e., each core i commits instructions in program order:

Property 1. $O_j \prec_{po} O_m \wedge (O_j, O_m \in \mathscr{O}^i) \Rightarrow e_j < e_m$ with $(e_j, e_m) \in \mathscr{R}^i_\oplus \times \mathscr{R}^i_\oplus \cup \mathscr{R}^i_\oplus \times \mathscr{W}^i_\oplus \cup \mathscr{W}^i_\oplus \times \mathscr{R}^i_\oplus \cup \mathscr{W}^i_\oplus \times \mathscr{W}^i_\oplus$.

Let us assume causality, i.e., the coherence actions in favor of a store can be launched only after it has been locally committed:

Property 2. $S_j \in \mathscr{S}^i_a \Rightarrow e_j < e'_j$, with e_j and e'_j conflicting at location a and $(e_j, e'_j) \in \mathscr{W}^i_\oplus \times \mathscr{W}^{x \neq i}_\odot$.

Properties 1 and 2 allow a checker to use commit events as anchors for efficiently checking completion orderings[12].

[12] Besides, Axiom 4 does not require full verification under Property 1, because there is no need for checking clauses 2, 3, and 4 (for $x = y$).

Let us also assume that the dynamic scheduler properly manages load and store buffers such that it preserves program order between a store and a conflicting load, i.e., the value of the store is always made available before a conflicting load could read it, formally[13]:

Property 3. $S_j \prec_{po} L_m \wedge (S_j, L_m \in \mathscr{O}^i_a) \Rightarrow e_j < e_m$, with e_j and e_m conflicting at location a and $(e_j, e_m) \in \mathscr{W}^i_{\oslash} \times \mathscr{R}^i_{\ominus}$.

Finally, let us assume that commit units properly handle outcomes, i.e., when a memory instruction is executed speculatively, its result is either committed or discarded:

Property 4. $\mathscr{R}^i_{\ominus} = R^i_{\ominus} \cup \overline{R}^i_{\ominus}$ and $R^i_{\ominus} \cap \overline{R}^i_{\ominus} = \emptyset$.

Property 5. $\mathscr{W}^i_{\oslash} = W^i_{\oslash} \cup \overline{W}^i_{\oslash}$ and $W^i_{\oslash} \cap \overline{W}^i_{\oslash} = \emptyset$.

Properties 4 and 5 allow runtime decisions while ruling out false diagnoses that would be induced by speculation.

Besides, we impose a couple of usual constraints (MANOVIT; HANGAL, 2006; HU et al., 2012; FREITAS; RAMBO; SANTOS, 2013) on test generation and execution. To keep a single set of monitors per core domain, we pragmatically constrain test execution, as follows:

Constraint 1. Each core runs a single thread.

Albeit it is not always possible to fully distinguish between operations by relying only on the events they induce, this distinction is required when tracking the order in which conflicting stores complete in different cores. To enable that, test generation must be enforced in such a way that the values assigned by distinct *conflicting* stores are unique, as formalized below[14]:

Constraint 2. $\forall S_j, S_m \in \mathscr{O}_a : Val[S_j] \neq Val[S_m] \neq Val^0_a$.

The next chapter experimentally evaluates checkers built upon the axioms, properties, and constraints formalized in this chapter.

[13] There is no need for checking Axiom 6 under Property 3.

[14] Indeed, this is less restrictive than in related work (e.g. Manovit & Hangal (2006), and Hu et al. (2012)), where each store is assigned a unique value, which serves as its identifier.

6 EXPERIMENTAL EVALUATION

This chapter experimentally evaluates the approach proposed in Chapter 5 within the framework described in Chapter 4. Section 6.1 specifies experimental conditions, Section 6.2 puts results in perspective, and Section 6.3 discusses details of the most relevant cases.

6.1 EXPERIMENTAL SETUP

We implemented two checkers for Section 5.1's axioms, one compliant with nMCA behaviors and another compliant with rMCA behaviors. Both verify Axioms 1 to 7, but the latter also verifies Axiom 9. We inserted monitors in the design representation according as described in Section 5.2, and built each checker under Section 5.3's assumptions. We compared them with the most recently reported runtime checker, which is based on multiple relaxed scoreboards (MSB) (FREITAS; RAMBO; SANTOS, 2013)[1].

We used gem5's infrastructure (BINKERT et al., 2011) to simulate design representations[2]. Since our approach is able to handle superscalar microarchitectures, we targeted ARMv8 (for which aggressive implementations exist), and we set up gem5 to simulate out-of-order cores (`O3CPU`). Besides, ARMv8 is among the best supported architectures on gem5. We relied on the `Ruby` submodule to simulate the memory system, because it is required for more elaborated cache protocols. For the interconnect network, however, we used `simple`. We adopted a two-level MOESI directory protocol, with 64KB (4-way) private caches at L1 and a shared 4MB (16-way) cache at L2, all operating with the same block size (64 bytes). These parameters were chosen to reflect aggressive ARMv8 implementations (e.g. ARM Cortex-A73).

To synthesize high-quality tests, we adopted the directing engine described in Chapter 4[3], and varied the number of shared locations in a test within a specified range (4, 8, 16, 32, 64, 128). We also constrained the generator to produce suites where all tests have fixed size, albeit distinct sizes were studied (1Ki, 2Ki, and 4Ki memory operations). We configured the RTG engine to use block-aligned effective addresses ($abc = 6$) and enforce true sharing ($sbc = true$). The choice of generation parameters was based on previous work (ANDRADE et al., 2018; ANDRADE; GRAF; SANTOS, 2020), except by the test sizes, which were chosen to best suit the limited time and resources available to simulation.

To quantify *false diagnoses*, we relied on designs containing no errors. Then we inserted different artificial errors to challenge the checkers by changing the FSMs that implement the coherence protocol (either by modifying the next state or precluding some due output action). Each faulty design contained a single, distinct error. The errors studied in the experiments

[1] We relied on the MSB implementation available from a previous research infrastructure to create a version compatible with our current framework.

[2] This choice was inspired by related works that also use gem5 in the experiments, such as (ELVER; NAGARAJAN, 2016), including our own previous works (FREITAS; RAMBO; SANTOS, 2013; ANDRADE et al., 2018; ANDRADE; GRAF; SANTOS, 2020; ANDRADE et al., 2020).

[3] Variant 2 was adopted for better trade-of between simulation time and detection rate.

Table 5 – Studied design errors.

ID	State(s)	Input event	Next state	Precluded output action
D1 (L1)	M	Fwd_GETS	M instead of O	(preserved)
D2 (L1)	O	Fwd_GETS	M instead of O	(preserved)
D3 (L1)	O	Load	M instead of O	(preserved)
D4 (L1)	O	Store	M instead of OM	(preserved as in (M, MM))
D5 (L1)	M	Fwd_GETX	I	data block in sendDataExclusive
D6 (L2)	ILXW	L1_WBDIRTYDATA	M	data block in writeDataToCache
D7 (L2)	ILOXW	L1_WBDIRTYDATA	M	data block in writeDataToCache
D8 (L2)	MI, OI	Writeback_Ack	I	data block in sendDataFromTBEToMemory
D9 (L2)	ILSW	L1_WBCLEANDATA	SLS	data block in writeDataToCache

Source: Graf et al. (2019)

are described in Table 5.

Historically, there is a difficulty to define a common reference for evaluating MCM checkers (HANGAL et al., 2004; MANOVIT; HANGAL, 2005; ROY et al., 2006; MANOVIT; HANGAL, 2006; SHACHAM et al., 2008; CHEN et al., 2009; HU et al., 2012; RAMBO; HENSCHEL; SANTOS, 2012; FREITAS; RAMBO; SANTOS, 2013; ELVER; NAGARAJAN, 2016; LEE; BERTACCO, 2017). To the best of our knowledge, there is no systematic methodology for synthesizing shared memory faults that can *statistically* represent actual design errors. Related works from industry obviously can not disclose details about errors found in the corporate environment. Academic works usually provide better (yet informal) descriptions. However, errors are bound to be tied to a specific MCM or protocol[4]. Since there is no error standardization in the research area, we were obliged to synthesize our own design errors to be compatible with the selected target (MCM and protocol). The required effort, of course, has limited the amount and variety of errors within the available time frame.

For a given test size, we launched the generator 12 times by exploiting different seeds[5], leading to 12 distinct test suites. We determined the fraction of them for which each checker raised errors in a correct design (i.e., false diagnoses). To determine the effort spent in an attempt to find a given error in a faulty design, we measured the runtime until the error was found or until generation was stopped, and we took the average on the set of all test suites. Runtimes were measured in an HP xw8600 workstation (Intel Xeon E5430, 2.66 GHz, 8 GB memory, Linux Mint 18.1 Serena). We obtained the overhead of our checker with respect to MSB by calculating the percentage of extra effort.

We evaluated the checkers over *nMCA designs* (those allowing nMCA behaviors) and *rMCA designs* (those *disallowing* nMCA behaviors). For rMCA designs, we relied on gem5's native representation. To obtain nMCA designs, we modified gem5's native (rMCA-compliant)

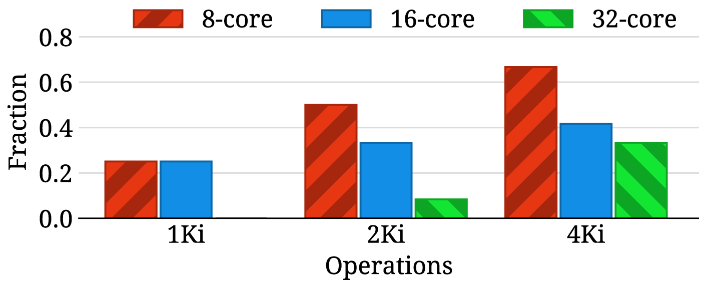
[4] For instance, Elver & Nagarajan (2016) describe errors specific to TSO and to their own TSO-CC protocol.
[5] The number of seeds was limited for compatibility with the available time frame.

coherence protocol, which was customized for allowing a core executing a store to forward its value to another core before it has received all invalidations[6] (GHARACHORLOO, 1995; TRIPPEL et al., 2017).

6.2 RESULTS IN PERSPECTIVE

Figure 14 shows the fraction of false diagnoses raised by MSB for distinct test sizes and growing core counts, when handling correct nMCA designs. For a given core count, the fraction of false diagnoses significantly increases with test size, which is inconvenient, because larger test sizes are usually required to expose the most subtle errors in faulty designs. As a result of properly modeling memory behavior, our checker did not raise any false diagnosis *at all* under exactly the same conditions.

Figure 14 – Fraction of false diagnoses raised by MSB for nMCA designs.



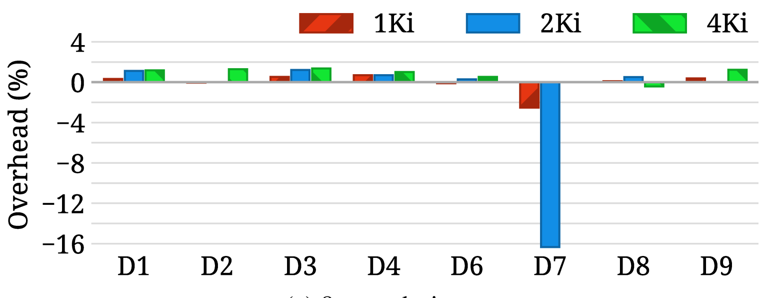
Source: Graf et al. (2019)

The reason behind the false diagnoses raised by the baseline checker is that it only observes *local* events (not *remote* events). Therefore, to verify the compliance between the value of a load and the value of the last observed conflicting store, MSB dynamically builds a global trace consisting of local write completion events and analyses that trace. When the design is strictly atomic, local and remote write completion events are indistinguishable, and the baseline checker works properly. However, under relaxed atomicity, a local write completion event happens before the respective remote write completion event, and a remote conflicting load may execute before the corresponding invalidation takes effect. In that case, the checker would interpret that the load has seen the old value of the location instead of the value of the last local write completion event, raising an error diagnosis. However, such behavior results from the intentional relaxation inherent to an nMCA design. Therefore, the detection of such apparent error leads to false diagnosis (false positive).

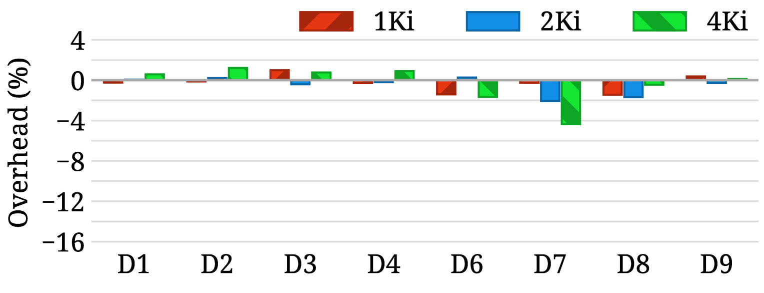
We compared our checker and MSB for detecting errors in faulty rMCA designs. Our checker was able to find all studied errors, but MSB was unable to find error D5. Figure 15 shows the average overhead of our checker with respect to MSB for the designs where both

[6] Due to the way the inter-cache buffers are implemented in gem5, the resulting design representation does not expose architecturally visible nMCA behaviors.

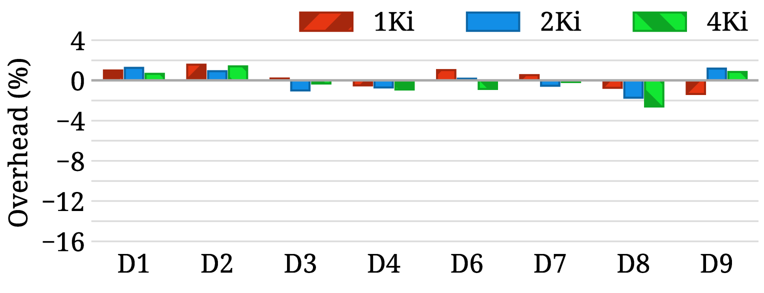
Figure 15 – Effort overhead for faulty rMCA designs.



(a) 8-core designs.



(b) 16-core designs.



(c) 32-core designs.

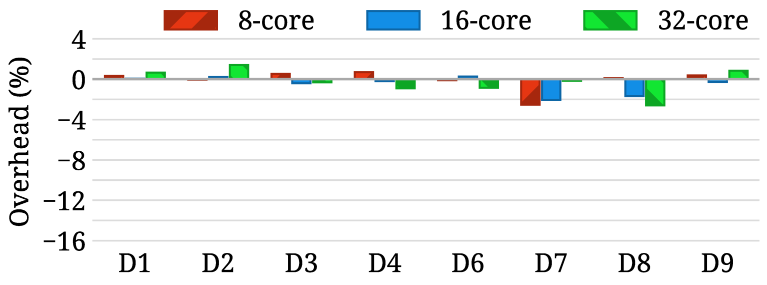
Source: Graf et al. (2019)

checkers exposed errors. The maximum effort overhead observed was 2.5%, whereas the maximum effort reduction was 16%. That is because our checker exploits redundancy. As opposed to the baseline, our checker not only verifies *values* but also the *ordering* of memory events. Since proper ordering is a necessary condition for proper values, improper ordering may lead to *early detection*. Figure 15 indicates that the approach's versatility and improved verification quality may come at the expense of negligible additional effort, but often leads to effort reduction.

Our checker happens to require more or less effort than the baseline checker depending not only on error type, but also on thread size, which decreases with growing core counts for a given test size. For better explaining such results, let us focus on tests with the same thread size. Figure 16 shows the average overhead of our checker with respect to MSB for tests with 128 operations per thread.

The effort required for finding an error is basically determined by simulation time, which is dominant over the time spent on checking axioms at runtime. Thus, the effort is

Figure 16 – Effort overhead for faulty rMCA designs and fixed thread size (128 operations).



Source: Graf et al. (2019)

largely determined by how many tests are needed to expose an error. In some cases, the higher number of monitors of our template happens to expose an error with less tests than required by the baseline checker. This explains the improvement in effort and why it can be non-negligible. However, when they happen to require the same amount of tests to expose an error, our checker needs slightly more effort, because it handles a larger number of events. The overhead in effort tends to be negligible, because both checkers handle essentially the same amount of local events for a given thread size, and the number of extra remote events handled by our checker would grow at most linearly with core count if every location was shared by each core, which is a rather unlikely worst-case scenario, especially under RTG.

6.3 DETAILED DISCUSSION OF MOST RELEVANT CASES

To explain why the proposed checker can be superior to the baseline checker in terms of error discovery rate and verification effort, let us now focus on a couple of cases from the results reported in the previous section.

Let us first explain why the baseline checker (MSB) was unable to detect error D5, which was found by the proposed checker. The reason is that the former does not observe memory events in remote cores, but only checks their impact on the locally observed values. To understand how this affects detection, consider the observability scenario for that error: imagine that the local core has just written to a block, when a remote core requests write permission for the same block. When it grants permission, the local core is supposed to send the contents of the block to the requester, but error D5 suppresses this last action. Under the conditions of the experiment, true sharing was enforced for all locations. In this case, the stores competing for the same block are certainly to the same location. Since the second store overwrites that location, the wrong value will not be observable by the MSB. Actually, the baseline checker would detect D5 if the competing writes were to different locations within the same memory block (i.e., if the true sharing constraint was relaxed). In contrast, our checker finds D5 regardless of memory allocation conditions. That is because it does not depend only on *value observation* to detect errors, but also on discovery of *improper orderings*.

Several causes concur to induce different detection behaviors depending on error type

Table 6 – Percentage of test suites leading to detection of D7 for 8-core designs.

	MSB	Speck&Check	
Test size	Improper value	Improper ordering	Improper value
1Ki (128/thread)	91.67%	50.00%	50.00%
2Ki (256/thread)	100.00%	66.67%	33.33%
4Ki (512/thread)	100.00%	50.00%	50.00%

Source: the author.

Table 7 – Percentage of test suites leading to detection of D7 for 16-core designs.

	MSB	Speck&Check	
Test size	Improper value	Improper ordering	Improper value
1Ki (64/thread)	91.67%	33.33%	66.67%
2Ki (128/thread)	100.00%	33.33%	66.67%
4Ki (256/thread)	100.00%	41.67%	58.33%

Source: the author.

Table 8 – Percentage of test suites leading to detection of D7 for 32-core designs.

	MSB	Speck&Check	
Test size	Improper value	Improper ordering	Improper value
1Ki (32/thread)	83.33%	8.33%	83.33%
2Ki (64/thread)	100.00%	0.00%	100.00%
4Ki (128/thread)	100.00%	16.67%	83.33%

Source: the author.

and experimental conditions. That is why detection behavior is easier to explain when one of them is clearly dominant. This is the case for the behavior observed for error D7. To explain the variation in the average effort overhead shown in Figure 15, Tables 6, 7 and 8 provide complementary information. They report the percentage of test suites leading to the exposure of error D7 with different detection mechanisms.

The variation of overhead for the same core count can be explained by the different impact of improper ordering as a detection mechanism for increasing *test* sizes. Let us focus on 8-core designs, for instance. The detection rate with the smallest test size (1Ki) was 100% for our checker, but 92% for the baseline checker. This indicates that improper ordering is a mechanism that can improve error detection, especially for short tests. For the two largest test sizes, the detection rate was 100% for both checkers, but our checker detected the error much earlier for the second largest test size (2Ki), because improper ordering was the dominant mechanism: 67% of the detection was due to improper ordering and 33% due to improper values. However, no early detection was observed for the largest test size, because improper ordering was not the dominant mechanism: 50% of the detection was due to improper ordering and 50% due to improper value. This indicates that improper ordering is a mechanism that can lead to early detection, especially for short tests, and consequently improve test throughput.

The variation of overhead with respect to growing core counts can be explained by the different impact of improper ordering as a detection mechanism for decreasing *thread* sizes. For a given test size, higher core counts make error detection more difficult. Let us focus on the shortest test size (1Ki), for instance. The detection rate of the baseline checker decreases from 92% to 83% as core count grows from 8 to 32. In contrast, the detection rate of our checker decreases from 100% to 92%. This indicates that improper ordering is a mechanism that makes detection less sensitive to growing core counts. However, improper ordering as a detection mechanism becomes less dominant: 50% of the detection was due to improper ordering for 8-core designs, 33% for 16-core designs, but only 8% for 32-core designs. That is because the number of operations per thread is reduced from 128 to 64 and then to 32 operations. The larger the thread, the higher the probability of conflicting stores, and the higher the probability of early detection due to improper ordering. The smaller the thread, the higher the probability of late detection due to improper values. Besides, detection due to improper values is more expensive for our checker as compared to the baseline due to the higher number of events. This explains why our checker (on average) leads to overhead, despite its higher detection rate.

Although errors D6 and D9 seem *structurally* similar to error D7, they do not show the same detection behavior. That is due to different controlability and observability requirements for defining the detection scenario of each error. For a given test suite, the detection of an error depends not only on the frequency in which the faulty transition is stimulated (*controlability*), but also on the frequency in which the suppressed action is observed (*observability*).

Let us first describe the observability scenario for error D7. Imagine that both the L1 and the L2 caches hold copies of the same memory block, but the L1 cache, being the owner, has the only valid dirty copy. When that block is about to be replaced, the L1 cache asks for write-back permission. When the L2 cache grants permission, it is supposed to update the block with the incoming data, but this action is suppressed by D7. Let us consider two scenarios. On the one hand, if a later load is the first operation to request that block, the wrong value will be observed, and the baseline checker will detect the error. On the other hand, if a later store is the first operation to request that block, the baseline checker will not detect the error, due to the conditions of the experiments, in which true sharing was enforced. That is, the store overwrites the value of that location, making the baseline checker unable to observe the wrong value and, therefore, unable to detect the error.

This means that, for the baseline checker, there is a single observability scenario (the first one). The rate of detection will depend on the frequency in which a randomly generated test induces that scenario. For 8-core designs and the smallest test size, the detection rate was 92%. For the largest test sizes, however, the detection rate was 100%, meaning that at least one instance of that single observability scenario was induced in each test suite. In contrast, our checker leads to 100% detection rate regardless of test sizes and memory allocation conditions, because it does not depend on improper values as the only detection mechanism. It can also detect the error due to improper ordering. In other words, both scenarios become observability scenarios.

In contrast, the efficiency of our checker is reduced for errors D6 and D9, because the baseline checker becomes as effective as ours for uncovering those errors. Albeit their similarity with error D7 ensures similar probability for the observability scenario, it does not ensure similar probability for the controlability scenario, which depends on stimulating different paths in the FSM. The baseline checker happens to be more efficient for errors D6 and D9 because the faulty transition is reached more often for D6 and D9 than it is for error D7. For instance, the faulty transition of error D6 is stimulated 100 times as often as D7's.

Overall, the discussed cases show that use of improper ordering as an additional detection mechanism not only makes our checker independent of memory allocation conditions, but it also can lead to early error detection. It also makes our checker less sensitive to growing core counts and allows the use of shorter tests to improve test throughput. Although it requires more monitors and, consequently, have to deal with more events, our checker shows negligible overhead in the cases where it does not lead to improvement.

7 CONCLUSIONS AND FUTURE WORK

Instead of relying on litmus test generation, whose coverage control is limited, this dissertation addressed shared memory verification under DTG. To raise the coverage of memory events, randomness was exploited: we adopted RTG as the basic test generation mechanism. To reduce the effort of reaching acceptable coverage at design time, feedback-loop control was adopted: we implemented DTG as coverage-directed RTG.

As opposed to litmus tests, random tests are not self-checking tests. That is why, under RTG, shared memory validation requires MCM checkers, which are based on the analysis of traces of memory operations monitored at each core domain. Such analysis is performed offline during post-silicon test or at runtime during pre-silicon verification.

If an MCM checker monitors only the interface with shared memory, this would lead to a *single memory trace per core*. However, shared memory validation based on a single trace per core is *intractable* (GIBBONS; KORACH, 1997). For this reason, post-silicon checkers exhibit *exponential behavior* with growing core counts, unless they sacrifice verification guarantees. That is why this dissertation addressed checkers using *multiple* monitors per core for *reducing the complexity* of the problem. This keeps verification scalable, hopefully without affecting verification guarantees.

However, to extend the number of monitors per core domain beyond the interface with the shared memory, extra monitors have to be inserted on entry to and on exit from cache buffers and pipeline buffers. As a result, such monitors may expose speculative behaviors and non-atomic store behaviors resulting from design optimizations (design artifacts), which are required by high-performance implementations of a given architecture. For this reason, this dissertation proposed an approach to the building of runtime checkers that are able to handle designs with speculative effects and non-atomic store behaviors.

The proposed abstract specification is *general enough* for building efficient checkers when targeting designs exposing either rMCA or nMCA behaviors. Our approach is largely independent of architecture (except for fences and a few other architecture-specific features), and it is largely independent of *micro*architecture, because the proposed observability template rely on monitors located at the interface with quite common structures. The experimental evidence indicates that a checker produced with our approach is effective, its overhead is negligible, it often reduces the effort to detect an error, and it does not raise false positive diagnoses when targeting a design with nMCA behaviors, as opposed to conventional checkers.

Therefore, this dissertation contributed to the building of a verification framework where test generation aims at high coverage with low effort, and runtime checking aims at error discovery with proper verification guarantees.

As future work, we intend to perform a more extensive experimentation to further evaluate the proposed approach. We expect to challenge our checkers with a higher diversity of design errors, under complementary experimental conditions, hopefully targeting coherence protocols other than MOESI.

Although the baseline checker was already compared against other consistency checkers reported in the literature (HENSCHEL; SANTOS, 2013), we intend to make a direct comparison with them within our framework (as described in Chapter 4). For instance, we plan to implement the runtime checker proposed by Shacham et al. (2008), whose code in the public domain is unfortunately not compatible with our infrastructure. Although that checker is older than the MSB, and it admittedly has limited verification guarantees, a comparison with our approach is likely to pay off, because it is probably the fastest checker. We also plan to compare our checkers with the post-silicon checker proposed by Hu et al. (2012). Such checker allows two modes of operation: one providing verification guarantees via backtracking, another providing no guarantees, because backtracking is disabled. Since the former is admittedly not scalable, we intend to demonstrate that even the latter is still inadequate to be used at design time, as a result of low error discovery rate and high effort. Unfortunately, its code is not available in the public domain, but the paper describes its algorithms, which we intend to implement within our infrastructure.

A consistency checker is considered complete if it can completely prove or disprove that any set of memory traces complies with an MCM (SHACHAM et al., 2008). The MSB checker (used as a baseline in our experiments) was proven complete for rMCA designs[1] (FREITAS; RAMBO; SANTOS, 2013). Since the proposed approach relied on more monitors per core than MSB, we raised the hypothesis that the checkers built under the proposed approach are also complete, and we are trying to prove it. Preliminary proofs (made by the author's advisor) indicate that the checker is complete for the architecture-independent axioms at least for update-based protocols. However, the proof for invalidate-based protocols is harder, because values cannot be used for labeling invalidate events. We recently updated the observability template for monitoring not only invalidate requests but also data replies. We believe that this allows the required labeling for completing the proof. Nevertheless, we postpone any claim on verification guarantees until the preliminary proof is extended and properly reviewed. Therefore, as future work, we intend to establish the theoretical guarantees achievable by our approach.

[1] In spite of that our checker was superior in terms of error discovery under true sharing.

BIBLIOGRAPHY

ABTS, D.; SCOTT, S.; LILJA, D. J. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In: **17th International Symposium on Parallel and Distributed Processing**. [S.l.]: IEEE Computer Society, 2003. ISSN 1530-2075.

ADIR, A. et al. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. **IEEE Design Test of Computers**, v. 21, n. 2, p. 84–93, mar. 2004. ISSN 0740-7475.

ADIR, A.; SHUREK, G. Generating concurrent test-programs with collisions for multi-processor verification. In: **7th IEEE International High-Level Design Validation and Test Workshop**. [S.l.]: IEEE, 2002. p. 77–82.

ADVE, S.; HILL, M. D. **Sufficient Conditions for Implementing the Data-Race-Free-1 Memory Model**. Madison, WI, 1992.

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

ALGLAVE, J. et al. GPU Concurrency: Weak Behaviours and Programming Assumptions. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 50, n. 4, p. 577–591, mar. 2015. ISSN 0362-1340.

ALGLAVE, J. et al. Fences in weak memory models. In: TOUILI, T.; COOK, B.; JACKSON, P. (Ed.). **Computer Aided Verification**. Berlin, Heidelberg: Springer-Verlag, 2010. p. 258–272. ISBN 978-3-642-14295-6.

ALGLAVE, J.; MARANGET, L.; TAUTSCHNIG, M. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. **ACM Transactions on Programming Languages and Systems**, Association for Computing Machinery, New York, NY, USA, v. 36, n. 2, jul. 2014. ISSN 0164-0925.

ANDRADE, G. A. G. et al. Steep Coverage-ascent Directed Test Generation for Shared-memory Verification of Multicore Chips. In: **2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. New York, NY, USA: Association for Computing Machinery, 2018. ISBN 978-1-4503-5950-4.

ANDRADE, G. A. G. et al. A Directed Test Generator for Shared-Memory Verification of Multicore Chip Designs. **accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, 2020.

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

ANDRADE, G. A. G.; GRAF, M.; SANTOS, L. C. V. dos. Chaining and Biasing: Test Generation Techniques for Shared-Memory Verification. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 39, n. 3, p. 728–741, mar. 2020. ISSN 1937-4151.

ARM. **Cortex-A9 MPCore, Programmer Advice Notice, Read-after-Read Hazards. ARM Reference 761319.** 2011. Disponível em: http://infocenter.arm.com/help/topic/com.arm.doc.uan0004a/UAN0004A_a9_read_read.pdf.

BARROSO, L. A. et al. RPM: a rapid prototyping engine for multiprocessor systems. **Computer**, IEEE, v. 28, n. 2, p. 26–34, feb. 1995. ISSN 1558-0814.

BINKERT, N. et al. The Gem5 Simulator. **SIGARCH Computer Architecture News**, Association for Computing Machinery, New York, NY, USA, v. 39, n. 2, p. 1–7, aug. 2011. ISSN 0163-5964.

CHATTERJEE, P.; SIVARAJ, H.; GOPALAKRISHNAN, G. Shared Memory Consistency Protocol Verification Against Weak Memory Models: Refinement via Model-Checking. In: BRINKSMA, E.; LARSEN, K. G. (Ed.). **14th International Conference on Computer Aided Verification**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. v. 2404, p. 123–136. ISBN 978-3-540-45657-5.

CHEN, K.; MALIK, S.; PATRA, P. Runtime Validation of Memory Ordering Using Constraint Graph Checking. In: **14th International Symposium on High Performance Computer Architecture**. [S.l.]: IEEE, 2008. p. 415–426. ISSN 2378-203X.

CHEN, Y. et al. Fast Complete Memory Consistency Verification. In: **15th International Symposium on High Performance Computer Architecture**. [S.l.]: IEEE, 2009. p. 381–392.

CLARKE, A. C. **Profiles of the Future: An Inquiry into the Limits of the Possible**. New York: Harper & Row, 1973. ISBN 978-0060107925.

COLLIER, W. W. **Reasoning About Parallel Architectures**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992. ISBN 0-13-767187-3.

DENNARD, R. H. et al. Design of Ion-Implanted MOSFET's with Very Amall Physical Dimensions. **IEEE Journal of Solid-State Circuits**, v. 9, n. 5, p. 256–268, oct. 1974. ISSN 1558-173X.

DEORIO, A.; WAGNER, I.; BERTACCO, V. Dacota: Post-silicon validation of the memory subsystem in multi-core designs. In: **15th International Symposium on High Performance Computer Architecture**. [S.l.]: IEEE, 2009. p. 405–416. ISSN 2378-203X.

DEVADAS, S. Toward a Coherent Multicore Memory Model. **Computer**, IEEE, v. 46, n. 10, p. 30–31, 2013.

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

ESMAEILZADEH, H. et al. Dark Silicon and the End of Multicore Scaling. In: IEEE. **38th Annual International Symposium on Computer Architecture**. [S.l.], 2011. p. 365–376.

ESMAEILZADEH, H. et al. Dark Silicon and the End of Multicore Scaling. **IEEE Micro**, IEEE, v. 32, n. 3, p. 122–134, 2012.

FINE, S.; FOURNIER, L.; ZIV, A. Using Bayesian networks and virtual coverage to hit hard-to-reach events. **International Journal on Software Tools for Technology Transfer**, v. 11, n. 4, p. 291–305, 10 2009. ISSN 1433-2787.

FINE, S.; ZIV, A. Coverage Directed Test Generation for Functional Verification Using Bayesian Networks. In: **40th Annual Design Automation Conference**. New York, NY, USA: ACM, 2003. p. 286–291. ISBN 1-58113-688-9.

FREITAS, L. S.; RAMBO, E. A.; SANTOS, L. C. V. dos. On-the-fly verification of memory consistency with concurrent relaxed scoreboards. In: **esign, Automation Test in Europe Conference Exhibition**. [S.l.]: IEEE, 2013. p. 631–636. ISBN 978-1-4503-2153-2. ISSN 1530-1591.

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

GHARACHORLOO, K. et al. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. **SIGARCH Computer Architecture News**, Association for Computing Machinery, New York, NY, USA, v. 18, n. 2SI, p. 15–26, may 1990. ISSN 0163-5964.

GIBBONS, P. B.; KORACH, E. Testing Shared Memories. **SIAM Journal on Computing**, Society for Industrial and Applied Mathematics, v. 26, n. 4, p. 1208–1244, aug. 1997. ISSN 0097-5397.

GOPALAKRISHNAN, G.; YANG, Y.; SIVARAJ, H. QB or Not QB: An Efficient Execution Verification Tool for Memory Orderings. In: ALUR, R.; PELED, D. A. (Ed.). **Computer Aided Verification**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 401–413. ISBN 978-3-540-27813-9.

GRAF, M. et al. Spec&Check: An Approach to the Building of Shared-Memory Runtime Checkers for Multicore Chip Design Verification. In: **International Conference on Computer-Aided Design**. [S.l.]: IEEE, 2019. p. 1–7. ISSN 1933-7760.

HANGAL, S. et al. TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model. **SIGARCH Computer Architecture News**, Association for Computing Machinery, New York, NY, USA, v. 32, n. 2, p. 114–123, mar. 2004. ISSN 0163-5964.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. 6th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN 0128119055.

HENSCHEL, O. P.; SANTOS, L. C. V. dos. Pre-silicon verification of multiprocessor SoCs: The case for on-the-fly coherence/consistency checking. In: **20th International Conference on Electronics, Circuits, and Systems**. [S.l.]: IEEE, 2013. p. 843–846.

HENZINGER, T. A.; QADEER, S.; RAJAMANI, S. K. Verifying sequential consistency on shared-memory multiprocessor systems. In: HALBWACHS, N.; PELED, D. (Ed.). **Computer Aided Verification**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. p. 301–315.

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

IBM. **POWER9 Processor User's Manual**. 2019. Disponível em: https://ibm.ent.box.com/s/tmklq90ze7aj8f4n32er1mu3sy9u8k3k.

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

LEE, D.; BERTACCO, V. MTraceCheck: Validating Non-Deterministic Behavior of Memory Consistency Models in Post-Silicon Validation. **SIGARCH Computer Architecture News**, Association for Computing Machinery, New York, NY, USA, v. 45, n. 2, p. 201–213, jun. 2017. ISSN 0163-5964.

LUSTIG, D. et al. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In: **International Conference on Architectural Support for Programming Languages and Operating Systems**. [S.l.]: Association for Computing Machinery, 2017. p. 661–675. ISBN 978-1-4503-4465-4.

LYU, Y. et al. Directed Test Generation for Validation of Cache Coherence Protocols. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 38, n. 1, p. 163–176, jan. 2019. ISSN 0278-0070.

MAMMO, B. W. et al. Post-Silicon Validation of Multiprocessor Memory Consistency. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 34, n. 6, p. 1027–1037, jun. 2015. ISSN 1937-4151.

MANOVIT, C.; HANGAL, S. Efficient Algorithms for Verifying Memory Consistency. In: **17th Symposium on Parallelism on Parallelism in Algorithms and Architectures**. New York, NY, USA: Association for Computing Machinery, 2005. p. 245–252. ISBN 1-58113-986-1.

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

MARCILIO, G. et al. A novel verification technique to uncover out-of-order DUV behaviors. In: **46th Annual Design Automation Conference**. New York, NY, USA: Association for Computing Machinery, 2009. p. 448–453. ISBN 9781605584973.

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

MOORE, G. E. Cramming More Components onto Integrated Circuits. **Electronics**, v. 38, n. 8, p. 114–117, apr. 1965.

PARK, S.; DILL, D. L. An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order). In: **Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures**. New York, NY, USA: Association for Computing Machinery, 1995. p. 34–41. ISBN 0897917170.

PULTE, C. et al. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. **Proceedings of the ACM on Programming Languages**, Association for Computing Machinery, New York, NY, USA, v. 2, n. POPL, dec. 2017.

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

RAMBO, E. A.; HENSCHEL, O. P.; SANTOS, L. C. V. dos. On ESL verification of memory consistency for system-on-chip multiprocessing. In: **Conference on Design, Automation, and Test in Europe**. [S.l.]: EDA Consortium, 2012. p. 9–14. ISSN 1530-1591.

ROY, A. et al. Fast and Generalized Polynomial Time Memory Consistency Verification. In: BALL, T.; JONES, R. B. (Ed.). **18th International Conference on Computer Aided Verification**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. v. 4144, p. 503–516. ISBN 978-3-540-37411-4.

SHACHAM, O. et al. Verification of Chip Multiprocessor Memory Systems Using a Relaxed Scoreboard. In: **41st IEEE/ACM International Symposium on Microarchitecture**. Los Alamitos, CA, USA: IEEE Computer Society, 2008. p. 294–305. ISBN 978-1-4244-2836-6.

TRIPPEL, C. et al. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. In: **22nd International Conference on Architectural Support for Programming Languages and Operating Systems**. New York, NY, USA: Association for Computing Machinery, 2017. p. 119–133. ISBN 978-1-4503-4465-4.

WAGNER, I.; BERTACCO, V. MCjammer: Adaptive Verification for Multi-core Designs. In: **Conference on Design, Automation, and Test in Europe**. New York, NY, USA: Association for Computing Machinery, 2008. p. 670–675. ISSN 1530-1591.

WATERMAN, A.; ASANOVI, K. **The RISC-V Instruction Set Manual Volume I: Unprivileged ISA**. [S.l.], 2019. Disponível em: https://riscv.org/specifications/.

ZHANG, M. et al. PVCoherence: Designing Flat Coherence Protocols for Scalable Verification. **IEEE Micro**, v. 35, n. 3, p. 84–91, may 2015. ISSN 0272-1732.