UNIVERSIDADE FEDERAL DE SANTA CATARINA
INFORMÁTICA E ESTATÍSTICA

Gabriel Arthur Gerber Andrade

EXPLOITING CANONICAL DEPENDENCE CHAINS AND ADDRESS BIASING CONSTRAINTS TO IMPROVE RANDOM TEST GENERATION FOR SHARED-MEMORY VERIFICATION

Florianópolis

2017

Gabriel Arthur Gerber Andrade

EXPLOITING CANONICAL DEPENDENCE CHAINS AND ADDRESS BIASING CONSTRAINTS TO IMPROVE RANDOM TEST GENERATION FOR SHARED-MEMORY VERIFICATION

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do Grau de Mestre em Ciência da Computação.
Universidade Federal de Santa Catarina
Orientador: Prof. Luiz Cláudio Villar dos Santos, Dr.

Florianópolis

2017

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

Andrade, Gabriel Arthur Gerber
Exploiting canonical dependence chains and address biasing constraints to improve random test generation for shared-memory verification / Gabriel Arthur Gerber Andrade ; orientador, Luiz Cláudio Villar dos Santos - Florianópolis, SC, 2017.
109 p.

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

Inclui referências

1. Ciência da Computação. 2. Verificação de projeto. 3. Memória compartilhada. 4. Coerência de memória. 5. 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.

Gabriel Arthur Gerber Andrade

EXPLOITING CANONICAL DEPENDENCE CHAINS AND ADDRESS BIASING CONSTRAINTS TO IMPROVE RANDOM TEST GENERATION FOR SHARED-MEMORY VERIFICATION

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

Florianópolis, 17 de Fevereiro de 2017.

Eu dedico este trabalho ao futuro.

ACKNOWLEDGEMENTS

Many thanks go to my advisor for his guidance, support, and confidence. His pragmatism has been crucial to finish this work in just two years. During this time, he has shown me how “less is more”.

I have been fortunate to be among great friends and colleagues. Particularly, I would like to thank the following people for providing the experimental infrastructure for my research: Eberle Andrey Rambo, Leandro da Silva Freitas, Olav Philipp Henschel, and Marleson Graf.

I would like to thank the members of the Examination Committee for their contributions to improve this work.

I would like to thank my parents, my relatives, and my friends. They were instrumental in supporting me. And, particularly, without these same people I would not be who I am today.

I would like to express my gratitude to the financial support of the Brazilian Research Agency CAPES (within the DS Program).

“Vamos em frente que atrás vem gente”
— A frase preferida de meus avôs.

Adolfo Gerdelmann Andrade e Guido Gerber (*in memoriam*).

RESUMO ESTENDIDO

Introdução
A verificação funcional do projeto de um sistema com multiprocessamento em *chip* (CMP) vem se tornando cada vez mais desafiadora por causa da crescente complexidade para suportar a abstração de memória compartilhada coerente, a qual provavelmente manterá seu papel crucial para multiprocessamento em *chip*, mesmo na escala de centenas de processadores. A verificação funcional baseia-se principalmente na geração de programas de teste aleatórios.

Trabalhos Correlatos e Gerador Proposto
Embora *frameworks* de verificação funcional que se baseiam na solução de problemas de satisfação de restrições possuam a vantagem de oferecer uma abordagem unificada para gerar estímulos aleatórios capazes de verificar todo o sistema, eles não são projetados para explorar não-determinismo, que é um importante mecanismo para expôr erros de memória compartilhada.
Esta dissertação reporta novas técnicas que se baseiam em lições aprendidas de ambos—os *frameworks* de verificação de propósitos gerais e as abordagens especializadas em verificar o modelo de memória. Elas exploram restrições sobre endereços e cadeias canônicas de dependência para melhorar a geração de testes aleatórios enquanto mantêm o papel crucial do não-determismo como um mecanismo-chave para a exposição de erros.

Geração de Sequências
Ao invés de selecionar instruções aleatoriamente, como faz uma técnica convencional, o gerador proposto seleciona instruções de acordo com cadeias de dependências pré-definidas que são comprovadamente *significativas* para preservar o modelo de memória sob verificação. Esta dissertação explora cadeias canônicas, definidas por Gharachorloo, para evitar a indução de instruções que, sendo desnecessárias para preservar o modelo de memória sob verificação, resultem na geração de testes ineficazes.

Assinalamento de Endereços
Em vez de selecionar aleatoriamente padrões binários para servir de endereços efetivos de memória, como faz um gerador convencional, o gerador proposto aceita restrições à formação desses endereços de forma a forçar o alinhamento de objetos em memória, evitar falso comparti-

lhamento entre variáveis e especificar o grau de competição de endereços por uma mesma linha de cache.

Avaliação Experimental
Um novo gerador, construído com as técnicas propostas, foi comparado com um gerador convencional de testes aleatórios. Ambos foram avaliados em arquiteturas de 8, 16, e 32 núcleos, ao sintetizar 1200 programas de testes distintos para verificar 5 projetos derivados, cada um contendo um diferente tipo de erro (6000 casos de uso por arquitetura). Os testes sintetizados exploraram uma ampla variedade de parâmetros de geração (5 tamanhos de programas, 4 quantidades de posições compartilhadas de memória, 4 *mixes* de instruções, e 15 sementes aleatórias). Os resultados experimentais mostram que, em comparação com um convencional, o novo gerador tende a expor erros para um maior número de configurações dos parâmetros: ele aumentou em 38% o potencial de expor erros de projeto. Pela análise dos resultados da verificação sobre todo o espectro de parâmetros, descobriu-se que os geradores requerem um número bastante distinto de posições de memória para alcançar sua melhor exposição. Os geradores foram comparados quando cada um explorou a quantidade de posições de memória correspondente à sua melhor exposição. Nestas condições, quando destinados a projetos com 32 núcleos através da exploração de todo o espectro de tamanhos de testes, o novo gerador expôs um tipo de erro tão frequentemente quanto a técnica convencional, dois tipos com 11% mais frequência, um tipo duas vezes, e um tipo 4 vezes mais frequentemente. Com os testes mais longos (64000 operações) ambos os geradores foram capazes de expor todos os tipos de erros, mas o novo gerador precisou de 1,5 a 15 vezes menor esforço para expor cada erro, exceto por um (para o qual uma degradação de 19% foi observada).

Conclusões e Perspectivas
Com base na avaliação realizada, conclui-se que, quando se escolhe um número suficientemente grande de variáveis compartilhadas como parâmetro, o gerador proposto requer programas de teste mais curtos para expor erros de projeto e, portanto, resulta em menor esforço, quando comparado a um gerador convencional.

Palavras-chave: Verificação de projeto. Memória compartilhada. Coerência de memória. Consistência de memória.

ABSTRACT

Albeit general functional processor verification frameworks relying on the solution of constraint satisfaction problems have the advantage of offering a unified approach for generating random stimuli to verify the whole system, they are not designed to exploit non-determinism, which is an important mechanism to expose shared-memory errors. This dissertation reports new techniques that build upon the lessons learned from both—the general verification frameworks and the approaches specifically targeting memory-model verification. They exploit address biasing constraints and canonical dependence chains to improve random test generation while keeping the crucial role of non-determinism as a key mechanism to error exposure. A new generator, built with the proposed techniques, was compared to a conventional random test generator. Both were evaluated for 8, 16, and 32-core architectures, when synthesizing 1200 distinct test programs for verifying 5 derivative designs containing each a different type of error (6000 use cases per architecture). The synthesized tests explored a wide variety of generation parameters (5 program sizes, 4 shared-location counts, 4 instruction mixes, and 15 random seeds). The experimental results show that, as compared to a conventional one, the new generator tends to expose errors for a larger number of parameter settings: it increased by 38% the potential for exposing design errors. By analyzing the verification outcomes over the full parameter ranges, we found out that the generators require quite distinct numbers of shared locations to reach best exposure. We compared them when each generator exploited the location count leading to its best exposure. In such conditions, when targeting 32-core designs by exploring the whole range of test lengths, the new generator exposed one type of error as often as the conventional technique, two types 11% more often, one type twice as often, and one type 4 times as often. With the longest tests (64000 operations) both generators were able to expose all types of errors, but the new generator required from 1.5 to 15 times less effort to expose each error, except for one (for which a degradation of 19% was observed).
Keywords: Design verification. Shared memory. Memory coherence. Memory consistency.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

FSM	Finite State Machine	28
CMP	Chip Multiprocessing	31
ARM	Advanced RISC Machine	38
RISC	Reduced Instruction Set Computing	38
CPU	Central Processing Unit	61
SPARC	Scalable Processor Architecture	61
MESI	Modified Exclusive Shared Invalid	61
LRU	Least Recently Used	61

LIST OF SYMBOLS

Test generation parameters

p	Number of processors	35
n	Number of memory operations	35
s	Number of shared locations	35
M	Target mix of canonical dependence chains	35
$seed$	Random seed	35

Memory operations

L	A Load operation	37
S	A Store operation	37
I_j	A load or store instruction	37
MB	A memory barrier	37
$\mathcal{O}$	Set of memory operations issued by all processors	37
$\mathcal{O}^i$	Set of memory operations issued by some processor i	37
$\mathcal{O}^i_a$	Subset of memory operations to the same location a	37
$(O_j)^i_a$	An operation O_j issued by processor i to location a.	37
Val^0_a	Initial value stored at location a	37
$Val[(O_j)^i_a]$	Value written or returned by some operation	37

Relations and dependence chains

$\prec_{po}$	Program order	37
$\leq$	Execution order	37
$\prec_{spo}$	Significant program order	39
$\leq_{co}$	Conflict order	39
$\prec$	Precedence relation	39
$\{A \prec B \prec\}_*$	Zero or more occurrences of a pattern	39
$\{A \prec B \prec\}_+$	One or more occurrences of a pattern	39
$\mathcal{C}_\kappa$	Set of chains from category κ in a given test program	40
C	Set of all chains	40

length(c)	Number of operations in chain c	40
length$^i(c)$	Number of operations in chain c issued by processor i	40
T^i	Thread assigned to processor i	42
$T^i[x]$	Content of the x-th slot of thread T^i	43
av^i	Number of available slots in thread T^i	43
$M[\kappa]$	Target proportion of chains specified for category κ	43
$C[\kappa]$	Fraction of all slots available for category κ	43
$Val[a]$	Last value written to location a	43
$\lambda_{min}(\kappa)$	Length feasibility of minimal chain from category κ	43
$\beta_{min}(\kappa)$	Breadth feasibility of minimal chain from category κ	43
$\phi_{min}(\kappa)$	Feasibility of minimal chain from category κ	43
X	Set of slots from thread T^i filled with operations	43
$X_{x>m}$	Subset of X with slots x such that $x > m$	43
max_a^i	Last slot of thread T^i referencing location a	43

Locations, addresses, and bit fields

A	Set of shared locations	35
a_i	A shared location in A	35
E	Set of effective addresses	35
e_i	An effective address in E	35
abc	Alignment biasing constraint	35
sbc	Sharing biasing constraint	35
cbc	Competition biasing constraint	35
κ	Number of addresses mapping to distinct cache rows	35
χ	Maximum degree of competition for a same cache row	35
C	Relation "locations competing for a cache row"	49
CP	Undirected graph representing a competition pattern	49
A_x	x-th component of a partition of A induced by C	50
N	Number of bits in an effective address	51
T	Number of bits in the tag field	51
I	Number of bits in the index field	51
O	Number of bits in the block offset field	51
α	Mapping from locations to effective addresses	52
$\mathcal{CP}_{\kappa,\chi,s}$	Set of CPs induced by $cbc = (\kappa, \chi)$ for s locations	52

$e.\mathcal{T}$	Tag field of address e	52
$e.\mathcal{I}$	Index of address e	52
$e.\mathcal{O}$	Block offset of address e	52
f	Inducer for a competition pattern	53
c	Cardinality of a connected component in a CP	53
CP_f	Competition pattern built from inducer f	53
f^*	Inducer for the primal competition pattern	54
σ	Current number of locations	54
$\mathcal{E}$	Available address space at a given iteration	55
i	Index i of some effective address	55
$\mathcal{E}_i$	Set of available addresses induced by index i	55
t	A tag t of some effective address	55
$\mathcal{E}_{i,t}$	Set of addresses induced by index i and tag t	55
$\mathcal{E}_i^{\mathcal{T}}$	Address subspace with same index i and different tags	56
$\mathcal{I}(A_x, sbc, \mathcal{E})$	Set of candidate indices	56

Metrics

(p, n, s)	A verification scenario	65
T	Collection of tests induced by a verification scenario	66
t_i	Measured runtime for test T_i	66
ε	Effectiveness of a verification scenario	66
$\hat{t}$	Average test runtime of a verification scenario	66
EF_G	Verification effort to expose error with generator G	66

SUMMARY

1 INTRODUCTION

Design verification has been challenged by the growing hardware complexity to support the *coherent* shared-memory abstraction, which is expected to keep its crucial role in Chip Multiprocessing (MARTIN; HILL; SORIN, 2012) even in the scale of hundreds of cores (DEVADAS, 2013), because general-purpose parallel programming requires implicit management of cache coherence. Besides, since memory operations might be allowed to execute out of program order, a *memory model* is provided as part of the programmer's view for reasoning about the ordering of memory operations across the multiple threads of a parallel program. It specifies *consistency* rules defining not only the degree of program order relaxation, but also the extent of store atomicity (ADVE; GHARACHORLOO, 1996). Due to the dominance of synchronized programs, consistency rules end up hidden from the ordinary programmer (HENNESSY; PATTERSON, 2011). Thus, weak memory models allow for higher performance without compromising programability.

As a result from such trends, two main difficulties challenge shared-memory verification: 1) the growing complexity due to cache coherence (the number of reachable states grows exponentially with core up-scaling); 2) the higher complexity due to memory consistency (weaker models lead to larger numbers of allowed behaviors).

Functional processor verification largely relies on random test program generation. For instance, the literature reports verification frameworks that cast test generation into a constraint satisfaction problem (LEWIN et al., 1995; BIN et al., 2002; ADIR; SHUREK, 2002; ADIR et al., 2004; NAVEH et al., 2007). Albeit such frameworks provide general, unified mechanisms to synthesize test programs that exercise the whole system, including the *shared memory subsystem*, they are not designed to exploit non-determinism, which is an important mechanism to expose shared-memory errors (HANGAL et al., 2004; ELVER; NAGARAJAN, 2016).

This explains the existence of verification approaches specifically targeting the shared memory. Formal methods (e.g. (CHATTERJEE; SIVARAJ; GOPALAKRISHNAN, 2002)) rely on model checking to prove coherence and consistency properties, but cannot handle the memory system in full detail. Simulation-based verification relies on the execution of *test programs* on a design representation of the memory system implementation (SHACHAM et al., 2008; FREITAS; RAMBO; SANTOS, 2013). Hybrid methods (e.g. (ABTS; SCOTT; LILJA, 2003)) combine formal and simulation-based approaches. The literature reports two complementary simulation-based approaches, as follows.

Protocol-based verification has the advantage of relying on a precise coverage metric. Since the finite state machine (FSM) of each cache controller is known for a given coherence protocol, it is possible to build the product FSM, which specifies all reachable states and transitions. Therefore, it is possible to track which states and transitions were covered by a test suite or to guide test program generation towards higher coverage. On the one hand, it allows a trade-off between coverage and test length by means of adaptive test generation (e.g. (WAGNER; BERTACCO, 2008)). On the other hand, test generation may exploit the decomposition of the state space into simpler structures to make full coverage viable (e.g. (QIN; MISHRA, 2012)). The main disadvantages of protocol-based verification are: it is not directly reusable across derivate designs with distinct protocols and it requires either the validation of the memory system in isolation or the use of an abstraction of the other parts of the full-system functional design (ELVER; NAGARAJAN, 2016).

Memory-model verification has the advantage of checking the expected behavior for the whole shared-memory system (since it captures coherence requirements and consistency rules). Its main disadvantage is the lack of a precise metric for coverage (since the memory model relies on an axiomatic specification of order relations, it cannot directly rely on a native FSM model, as a coherence protocol does). The fact of not being tied to a specific protocol, however, leads to its main advantage: its direct reusability across derivative designs. On the one hand, it allows adaptive test-directed generation by selecting memory operations contributing to increase the non-determinism of a test (e.g. (ELVER; NAGARAJAN, 2016)). On the other hand, it allows the reuse of techniques originally developed for post-silicon checkers (e.g. (HANGAL et al., 2004)), which are based on constrained random test generation.

This dissertation reports new techniques for random test generation that build upon the lessons learned from both approaches — the general verification frameworks based on constraint satisfaction (LEWIN et al., 1995; BIN et al., 2002; ADIR et al., 2004; NAVEH et al., 2007) and the methods specifically targeting memory-model checking (HANGAL et al., 2004; MANOVIT; HANGAL, 2006; SHACHAM et al., 2008; HU et al., 2012), as follows:

- *Graph-based random address assignment*: From the former, it borrows the notion of address biasing constraints (ADIR et al., 2004), which are important mechanisms to induce interesting behaviors such as cache eviction. The dissertation proposes a new graph-based mechanism to exploit them when specifically targeting memory-model verification.

- *Chain-based random sequence generation*: From the latter, it preserves the exploitation of non-determinism, because aggressive data races tend to expose multiprocessor bugs faster (MANOVIT; HANGAL, 2006), as has been observed in both industrial (HANGAL et al., 2004) and academic (SHACHAM et al., 2008) environments. Typically, a test program generator offers control parameters such as frequency of instruction types, number of memory operations, and number of shared locations. Besides, a generator generally allows users to specify desirable sequences of memory operations to exercise known corner cases (HANGAL et al., 2004). Instead of simply allowing the *ad hoc* specification of desirable sequences, the dissertation proposes a novel mechanism that builds upon a formal specification (GHARACHORLOO, 1995) to automatically select the proper sequences (ANDRADE; GRAF; SANTOS, 2016).

Instead of directly solving a unified constraint satisfaction problem for the whole system, we deliberately reduce the scope of verification to the shared-memory subsystem and we decompose it into two main subproblems: constrained random sequence generation and constrained random address assignment. Such decomposition was induced by two properties of canonical dependence chains (GHARACHORLOO, 1995): their ability to rule out orderings not required to preserve the semantics of a memory model and their independence from the actual effective addresses assigned to the memory locations that are shared by the operations of a chain. That decomposition allows the exploitation of *significant operation orderings* as constraints (so as to avoid the generation of sequences unable to expose errors) and the exploitation of distinct *address patterns* for a given collection of sequences (so as to allow control on cache eviction as another key mechanism to error exposure). The reduction in scope and the decomposition fostered the design of specific algorithms to solve the subproblems instead of relying on generic solvers.

The remainder of this dissertation is organized as follows. The next chapter summarizes related work and informally presents the decomposition of the target problem by showing an overview of the interacting engines solving distinct subproblems and the generation flow underlying the proposed technique. Chapters 3, 4, and 5 formalize the target subproblems and describe the algorithms solving them. Chapter 6 compares a new generator (built with the proposed techniques) with a typical random generator in terms of their potential to error exposure, their effectiveness in exposing errors, and their impact on verification effort. Chapter 7 draws the main conclusions and discusses current limitations and future work.

A significant part of the text of this dissertation reflects the contents of other documents written in co-authorship: a published paper (ANDRADE; GRAF; SANTOS, 2016) and an article in preparation for submission, which were the result of strongly collaborative research. The author acknowledges the contributions of the co-authors of those documents for the joint research effort and points out his exclusive contributions to that collaborative effort, which are the object of this dissertation: the development of the algorithms solving Problems 1, 2, and 3; their implementation, and their experimental validation and evaluation.

2 RELATED WORK AND PROPOSED GENERATOR

This chapter briefly reviews related works on checkers and test generators for shared-memory verification so as to propose, in face of them, a new generator.

2.1 POST-SILICON CHECKERS

Memory-model checkers changed the approach to the problem of memory verification in Chip Multiprocessing (CMP) systems. Instead of directly dealing with the implementation, they exploit a pre-existing programmer's view abstraction, the memory model, for reducing the coupling between verification and implementation details. The seminal paper by Hangal, Vahia, and Manovit (HANGAL et al., 2004) inspired many post-silicon checkers (e.g. (MANOVIT; HANGAL, 2006; ROY et al., 2006; CHEN et al., 2009; HU et al., 2012)), which elaborated on their original idea. This allowed for more reusable checkers and extended post-silicon testing (the backbone of processor design validation efforts) beyond race-free self-checking tests and towards more effective pseudorandom tests with intensive data races. Despite suggestions (HANGAL et al., 2004; HU et al., 2012) that post-silicon approaches could be efficiently reused as pre-silicon checkers, they only hold for best effort versions of post-silicon checkers, but not for the complete versions providing verification guarantees for each test (MANOVIT; HANGAL, 2006; HU et al., 2012). In the latter case, post-silicon checkers exhibit poor scalability with the number of cores, which tends to severely limit their use at design time.

Most post-silicon checkers are postmortem engines, i.e. they require all traces to be available before starting their verification. Most of them rely on directed acyclic graphs to model order relations inferred from traces and, consequently, they prove memory incorrectness by counterexample. However, albeit the detection of a cycle is proof of incorrectness, the non-detection of any cycle is not proof of correctness, because some order relation might not have been inferred by the traces.

2.2 PRE-SILICON CHECKERS

A single pre-silicon *postmortem* checker is reported in the literature (RAMBO; HENSCHEL; SANTOS, 2012). To rule out false negatives, it exploits the extra observability of a design representation by oversampling memory events from two sequences per processor so as to infer order relations with extended bipartite graph matching.

Two classes of pre-silicon *runtime* checkers are reported in the literature. 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 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, the technique admittedly may raise false negatives for a given test program. In contrast, a recent work (FREITAS; RAMBO; SANTOS, 2013) proposes the use of multiple verification engines (one per core) and a single global checker to build an axiom-based on-the-fly checker. An error is raised as soon as it is found either by a local engine or by the global checker. The checker offers proven guarantees for a given test program (neither false positives nor false negatives if an error is exposed by the program).

2.3 RANDOM TEST GENERATION

Industrial environments have been relying on random stimuli generators for the functional verification of processors since the mid-1980 (AHARON et al., 1991). In the next decade, IBM developed the first model-based pseudorandom test generator. In the early 2000's, test generators have relied on more powerful engines, which translate the test generation problem into a constraint satisfaction problem and use a solver customized for pseudorandom test generation. Such approaches have the advantage of offering a unified framework for generating stimuli to verify the whole system (processor, memory, and interconnect). The downside of such unifying approach is that it makes it more difficult to exploit non-determinism (which is an important mechanism to expose shared-memory errors). Although a recent paper (ADIR et al., 2014) shows an extension to handle non-determinism within such verification methodology, it admittedly does not address the memory subsystem (but only the verification of the instruction set support for transactional memory).

On the other hand, it has been observed in both academic and industrial environments that non-determinism is an important key to shared-memory verification: programs with intensive data races expose bugs faster (HANGAL et al., 2004). As opposed to general, unifying verification methodologies, this observation has fostered specific pseudorandom test generation techniques targeting the memory subsystem.

Random tests targeting memory-model checking have been used by post-silicon checkers (HANGAL et al., 2004; ROY et al., 2006; MANOVIT; HANGAL, 2006; HU et al., 2012) in industrial environments and by pre-silicon checkers (SHACHAM et al., 2008; FREITAS; RAMBO; SANTOS, 2013) in academic environments. Most works focus on describing the analysis algorithms, albeit a brief description of the generation parameters is sometimes reported (HANGAL et al., 2004) or can be inferred (SHACHAM et al., 2008). Except for two works (RAMBO; HENSCHEL; SANTOS, 2011; ANDRADE; GRAF; SANTOS, 2016), no pseudocode for a typical generator could be found. Common parameters are the frequency of instruction types (HANGAL et al., 2004; MANOVIT; HANGAL, 2006; HU et al., 2012), the number of shared locations (HANGAL et al., 2004; MANOVIT; HANGAL, 2006; SHACHAM et al., 2008), and even desirable sequences to induce known corner cases. Instead of simply allowing the *ad hoc* specification of desirable sequences, a recent work (ANDRADE; GRAF; SANTOS, 2016) builds upon a formal specificion (GHARACHORLOO, 1995) to automatically select the proper sequences.

2.4 TEST-DIRECTED GENERATION

Since the hardware's throughput is orders of magnitude higher than a simulator's, pre-silicon verification can neither afford long tests nor large test suites to achieve coverage goals, as opposed to post-silicon testing. That is why test-directed generation has been advocated (WAGNER; BERTACCO, 2008) (QIN; MISHRA, 2012) (ELVER; NAGARAJAN, 2016) to bridge the coverage gap that would be induced by the random generation of shorter tests and smaller test suites when targeting protocol-based and memory model verification.

In face of core up-scaling, one of the keys to scalability is the decomposition of the state space. 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). Cooperating agents formulate their coverage goals in terms of the dichotomic FSM, not the product FSM. Another technique (QIN; MISHRA, 2012) decomposes the state space into simpler structures such as hypercubes and cliques, which can be traversed (in an Euler tour) to avoid visiting the same transitions many times. It may allow full coverage with tests 50% shorter than a breadth-first traversal.

To maximize coverage without increasing test length, adaptive test generation is preconized. For instance, based on the coverage goals attained by the tests already generated, a MCjammer's agent makes a probabilistic decision to either pursue its own goal, or to generate a stream of instructions to allow another agent to achieve its goal, or to execute a *random stream of memory accesses* (WAGNER; BERTACCO, 2008). In McVerSi (ELVER; NAGARAJAN, 2016), a genetic programming approach is used to progressively improve the quality of the test suite. It relies on a crossover function that prioritizes memory operations contributing to non-determinism, thereby increasing the probability of uncovering coherence and consistency errors.

Albeit adaptive test-directed generation may lead to high coverage for both protocol-based and memory model verification, it has to rely on some basic random generation engine to expose new frontiers for improvement. Therefore, random test generation can be exploited as a standalone approach or as part of an adaptive test-directed generation framework.

As shown in an early work (ANDRADE; GRAF; SANTOS, 2016), there is unexploited room for improvement by creating a technique lying in between random and directed test generation. This dissertation generalizes such early work, as shown in the next section.

2.5 AN OVERVIEW OF THE PROPOSED GENERATOR

As shown in Figure 1, the proposed generator consists of three interacting engines: a sequence generator, an address assigner, and an instruction synthesizer.

Given the number of processors (p) of the architecture, the target numbers of memory operations (n) and shared locations (s) for the test program, a target mix of canonical dependence chains, and a random seed, the *sequence generator* builds p random sequences containing each n/p operations with references to locations in the set $A = \{a_1, a_2, \cdots, a_s\}$. To build the sequences, the generator exploits

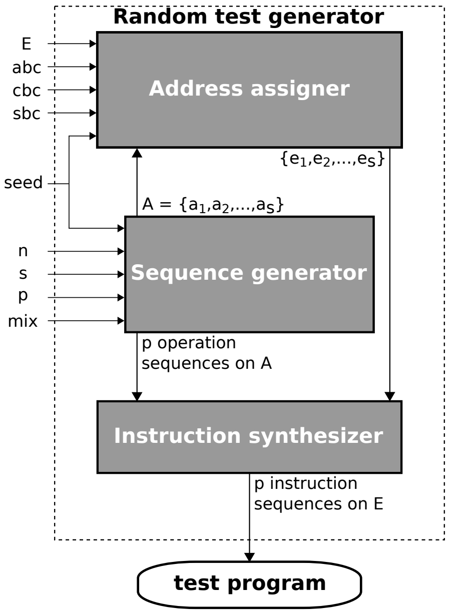


Figure 1: Structure of the proposed generator

dependence chains so as to increase the probability of error exposure (ANDRADE; GRAF; SANTOS, 2016) by relying on canonical chains that provenly preserve the semantics of the original memory model (GHARACHORLOO, 1995). Given the set of effective addresses (E) defined by the address space of a given architecture, the address assigner maps *locations* $a_1, a_2, \cdots, a_s$ to effective *addresses* $e_1, e_2, \cdots, e_s$. The mapping relies on three types of biasing constraints to enforce desirable properties for the effective addresses to be assigned:

- *Alignment biasing constraint* (*abc*): a natural number specifying that the address is aligned to 2^{abc} bytes.
- *Sharing biasing constraint* (*sbc*): a Boolean value enforcing true sharing or not.
- *Competition biasing constraint* (*cbc*): a pair (κ, χ), with $\kappa, \chi \in \{1, 2, \cdots, s\}$, indicating the number of addresses mapping to distinct cache rows and the maximum degree of competition for a same cache row (as will be explained later).

The *instruction synthesizer* converts p sequences of memory *operations* referencing locations into p sequences of memory *instructions* referencing effective addresses. Besides, it also defines the values to be written by store instructions.

The next three chapters formalize the problems addressed by each engine and propose algorithms to solve them.

3 SEQUENCE GENERATION

This chapter describes how canonical dependence chains are exploited when generating sequences of operations. First, it illustrates the key ideas and formalizes the notions required to formulate the target problem. Then it explains how the generator works by means of an example. Finally, it describes the algorithm proposed to solve the constrained random sequence generation problem.

3.1 NOTATION

We rely on the following notation. $\mathcal{O}$ is the set of memory operations issued by all processors. $\mathcal{O}^i \subset \mathcal{O}$ denotes the operations induced by the instructions issued by some processor i. We let p denote the total number of processors. Therefore, $\mathcal{O} = \cup_{i=1}^{p} \mathcal{O}^i$. A is the set of all locations referenced by operations in $\mathcal{O}$. $\mathcal{O}_a^i \subset \mathcal{O}^i$ denotes the subset of operations to the same location $a \in A$. To specify that an operation O_j is issued by a processor i and makes a reference to a location $a \in A$, we write $(O_j)_a^i$. We replace O by L or S to specify that the operation is either a load or a store. In shorthand notation, we may drop one of the subscripts or the superscript when irrelevant. Besides, we let Val_a^0 denote the initial value stored at location a before any processor ever writes to it. Finally, let $Val[(O_j)_a^i]$ be the value written or returned by some operation issued by processor i.

Given two (load or store) instructions, say I_j and I_m, if I_j precedes I_m in some thread, their respective memory operations are in program order, written $O_j \prec_{po} O_m$. Therefore, $\prec_{po}$ is a partial order on the set of all memory operations and a total order on the set of operations of a given thread.

A test program may induce many executions with distinct outcomes. An execution induces a memory *behavior*. Every valid behavior of a memory system must satisfy a partial order $\leq$ on the set of memory operations.

From the *program order* $\prec_{po}$, a memory model specifies the allowed *execution orders* by means of a partial order $\leq$, which is formally defined by axioms. Examples of such axioms can be found in the literature for distinct models (MANOVIT; HANGAL, 2006) (FREITAS; RAMBO; SANTOS, 2013) (ROY et al., 2006).

For convenience, but without loss of generality, we selected as target the memory model adopted in the Alpha architecture (GHARACHORLOO, 1995), whose weak enforcement of program order leads to a larger number of valid states as compared to less relaxed models such as the Total Store Ordering model (GHARACHORLOO, 1995) for instance. Besides, the target model closely ressembles the popular Weak Ordering model adopted in ARMv7.

For simplicity, we avoid an axiomatic description of the target memory model, since it is not required for understanding the proposed technique. Essentially, the adopted target model allows loads and stores to different locations to complete out of program order unless there is a memory barrier between them (GHARACHORLOO, 1995). Such an informal description of the memory model is sufficient for the formulation of dependence chains, which are addressed in the next section.

3.2 KEY IDEAS

We were aware that Gharachorloo had formalized aggressive specifications that provenly preserve the semantics of memory models (GHARACHORLOO, 1995). His conditions aimed to enable the *design* of an aggressive implementation that provides higher performance than the original model but leads to the same results. We realized that we could exploit those conditions for a different use: the constrained pseudorandom generation of test programs for *verification*.

Our insight was the following. A test program should avoid inducing operation orderings that are unnecessary for maintaining the semantics of the memory model under verification, since it will not expose errors. Therefore, to rule out the generation of ineffective test programs, we should induce only *significant operation orderings*. It turns out that, among the various conditions of an aggressive specification, Gharachorloo specifies the actual significant orderings in the form of uniprocessor and multiprocessor dependence chains.

Instead of randomly selecting instructions as typical generators do, we select instructions according to predefined dependence chains that are provenly significant for keeping memory model semantics. Since the ordering between some of the elements of a chain depends on their execution order, despite using proper verification patterns, the generated test program may not always induce a given chain at runtime. When this happens, some ineffective orderings may remain in the test, as if they were induced by a conventional generator.

3.3 MAIN NOTIONS

Definition 1 *Let $O_j \in \mathcal{O}_a^i$ and $O_m \in \mathcal{O}_b^i$. Let MB be a memory barrier, i.e. a mechanism to restore program order between load and store operations whose order is relaxed by the memory model*[1]*. We say that two operations are in significant program order, written $O_j \prec_{spo} O_m$, iff one of the following holds: $(O_j)_a \prec_{po} (O_m)_{b=a}$ or $(O_j)_a \prec_{po} MB \prec_{po} (O_m)_{b \neq a}$.*

Definition 2 *We say that two operations are in conflict order, written $O_j \leq_{co} O_m$, iff $(O_j)_a^i \leq (O_m)_a^k$ and at least one of them is a store.*

Definition 3 *We say that two operations are in significant conflict order, written $O_j \leq_{sco} O_m$, iff $(L_j)_a^i \leq (S_m)_a^k \vee (S_j)_a^i \leq (L_m)_a^k \vee (S_j)_a^i \leq (S_m)_a^k \vee (L_j)_a^i \leq S_a^x \leq (L_m)_a^k$.*

Definition 4 *A chain is a sequence $X \prec A \prec \cdots \prec B \prec Y$, where the endpoints X and Y are memory operations, but $A, \cdots, B$ may represent either memory operations or memory barriers. The precedence relation $\prec$ between two successive elements denotes one of the relations $\prec_{po}$, $\prec_{spo}$, $\leq$, $\leq_{co}$, or $\leq_{sco}$. For convenience, let $\{A \prec B \prec\}_*$ denote zero or more occurrences of this pattern in the chain. Similarly, let $\{A \prec B \prec\}_+$ denote one or more occurrences of the pattern.*

The above formalized notions allows us to formulate the chains to be used by our generator.

3.4 CHAIN CATEGORIES

A dependence chain is one that may induce data dependency through memory, either within the scope of the same thread (uniprocessor chain) or across threads allocated in distinct processors (multiprocessor chain). Gharachorloo (GHARACHORLOO, 1995) has defined a single category of significant uniprocessor dependence chain and three

[1]In the Alpha processor, there are two flavors of instructions providing memory barriers: MB (which may intervene between arbitrary memory instructions) and WMB (which may intervene between store instructions). In the latter case, an extra clause would be required in Definition 1: $(O_j)_a \prec_{po} WMB \prec_{po} (O_m)_{b \neq a}$ and O_j and O_m are stores. We prefer to omit such technicality to keep the definition independent from the choice of instruction-set architecture.

categories of multiprocessor dependence chains. Each category of significant chain is defined below. To simplify the notation, the defining patterns use L, S, O to denote *types* of memory operations (respectively, load, store, any). We implicitly assume that distinct operation instances of each type will be used for actually building the chain.

Category 0: $O_a^i \prec_{po} \{O_a^i \prec_{po} O_a^i \prec_{po}\}_* \, O_a^i$, where two successive elements cannot be of load type.

Category 1: $S_a^i \leq L_a^j \prec_{po} L_a^j$ or $S_a^i \leq L_a^j \prec_{po} S_a^j$, where $i, j \in \{1, \cdots, p\}$ and $i \neq j$.

Category 2: $O_a^i \prec_{spo} \{O_b^i \leq_{sco} O_b^j \prec_{spo}\}_+ \, O_a^j$, where $i, j \in \{1, \cdots, p\}$, $i \neq j$ and b is arbitrary.

Category 3: $S_a^i \leq_{sco} L_a^j \prec_{spo} \{O_b^j \leq_{sco} O_b^k \prec_{spo}\}_+ \, L_a^k$, where $i, j, k \in \{1, \cdots, p\}$, $i \neq j$, $j \neq k$ and b is arbitrary.

3.5 PROBLEM FORMULATION

Our generator accepts four main parameters: the total number of memory operations in the test program (n), the total number of processors (p), the number of distinct shared locations (s), and the target mix of patterns (M). The mix specifies the target fraction of chains from each category. Let $\mathcal{C}_\kappa$ denote the set of chains from category κ in a given test program and let $\mathcal{C} = \cup_{\kappa=0}^{3} \mathcal{C}_\kappa$ be the set of all chains. Let length(c) be the number of operations in chain c and let length^i(c) be the amount of them that are issued by processor i. The addressed target problem can be formulated as follows:

Problem 1 *Given n, p, s, M, find a set of chains $\mathcal{C}$ subject to the following constraints:*

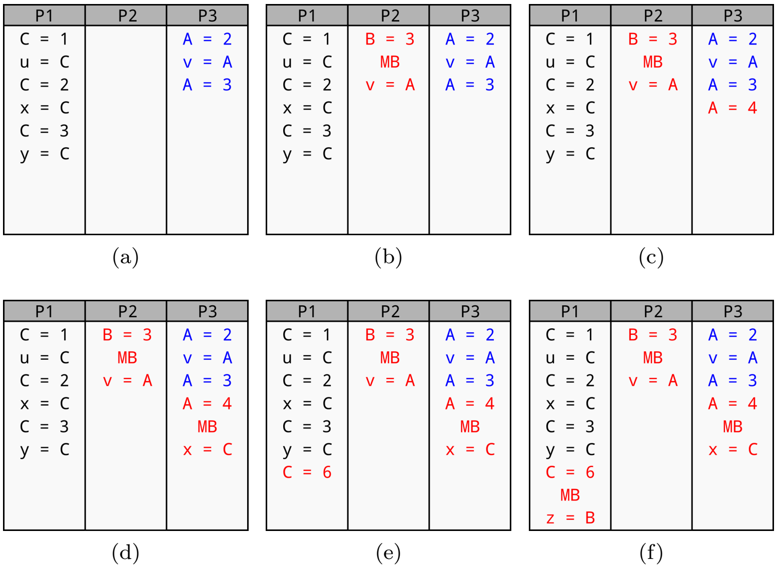
- $\sum_{c \in \mathcal{C}}$ *length* $(c) = n$,
- $\sum_{c \in \mathcal{C}}$ *length*$^i(c) = n/p$ *for all* $i \in \{1, \cdots, p\}$,
- $a \in \{a_1, a_2, \cdots, a_s\}$ *for each memory operation* O_a *in* $\mathcal{C}$,
- $|\mathcal{C}_\kappa| / |\mathcal{C}| = M[\kappa]$ *for each* $\kappa = 0, 1, 2, 3$.

The first constraint specifies that all memory operations are part of a chain; the second, that all threads have the same number of memory operations; the third, that all operations must use one of the shared locations; the forth, that the obtained proportions should reach the target mix.

3.6 AN EXAMPLE

The proposed generator assumes that a single thread is assigned to each processor and all threads have the same number of slots for memory operations. For simplicity, the same latency is assumed for all memory operations. Therefore, successive slots from distinct processors are likely to reflect execution order as far as that hypothesis holds (this simplifies the generation process, but it does not preclude further elaborations from enforcing the execution order of operations belonging to distinct threads).

The example in Figure 2 shows how the proposed generator inserts a chain from category 2 in a test program under construction. Uppercase identifiers denote variables in memory; lowercase identifiers, variables in registers. Memory barriers are labeled as MB. Suppose that two chains from category 0 (uniprocessor dependence chain) were previously created (Figure 2a), one in processor P1 (black), another in processor P3 (blue).



P1	P2	P3
C = 1		A = 2
u = C		v = A
C = 2		A = 3
x = C		
C = 3		
y = C		

(a)

P1	P2	P3
C = 1	B = 3	A = 2
u = C	MB	v = A
C = 2	v = A	A = 3
x = C		
C = 3		
y = C		

(b)

P1	P2	P3
C = 1	B = 3	A = 2
u = C	MB	v = A
C = 2	v = A	A = 3
x = C		A = 4
C = 3		
y = C		

(c)

P1	P2	P3
C = 1	B = 3	A = 2
u = C	MB	v = A
C = 2	v = A	A = 3
x = C		A = 4
C = 3		MB
y = C		x = C

(d)

P1	P2	P3
C = 1	B = 3	A = 2
u = C	MB	v = A
C = 2	v = A	A = 3
x = C		A = 4
C = 3		MB
y = C		x = C
C = 6		

(e)

P1	P2	P3
C = 1	B = 3	A = 2
u = C	MB	v = A
C = 2	v = A	A = 3
x = C		A = 4
C = 3		MB
y = C		x = C
C = 6		
MB		
z = B		

(f)

Figure 2: An example of how to enforce a significant ordering

Suppose that category 2 was randomly selected as a chain target. The generator first tries to build a minimal chain from the selected category, but might randomly decide to extend it as far as it does not violate any constraints. To build the minimal chain, the generator randomly selects a processor (say P2) and adds two memory operations with an intervening memory barrier to the first free slots of the respective thread (Figure 2b). The memory operation types are randomly selected (in Figure 2b, the first one turned out to be a store; the second, a load). Then a new processor is randomly selected (say P3) for the next operations of that chain. To comply with category 2 specifications, the first of them must conflict with the previous operation already in the chain and, thus, neither its type nor its location can be randomly selected (in Figure 2c, it *must* be a store to location A).

At this stage, the generator randomly decides whether the chain will be kept minimal or be extended (in the former case, the next memory operation becomes an endpoint and must conflict with the chain's starting point). Assume that the generator decided for extension. In this case, the location of the next operation is randomly selected (say C), because that operation is not intended anymore as an endpoint (Figure 2d).

To accomplish the extension, a processor is randomly selected (say P1) and an operation conflicting with the previous one in the chain is inserted (Figure 2e). Again, the generator must decide whether the chain should be further extended or not. Suppose that, this time, the random decision is for no further extension. As a result, the last operation, being an endpoint, cannot be randomly selected, since it must conflict with the chain's starting point (in Figure 2f, that endpoint *must* be a load from location B).

The memory operations in red represent the resulting chain. The significant conflict ordered required for actually forming that chain in runtime will only take place if the order implied by the slots turns out to be the actual execution order. In spite of that, the example shows that, instead of allowing fully random generation of operation sequences, the proposed technique constrains the sequences to comply, as much as possible, with significant orderings.

3.7 PSEUDOCODE

Let us formalize a few notions on which our algorithms rely. Let T^i denote the thread assigned to processor i. Each thread consists of

n/p slots. We write $T^i[x]$ to denote the content of the x-th slot of a thread. All slots are empty before generation is launched. During generation, operations are assigned to empty slots in each thread. Our algorithms track the *number of available slots* in thread T^i, which is denoted as av^i. Operations can be selected from three different *types*: load (L), store (S) or memory barrier (MB). To cope with the target proportion of chains specified for some category κ, i.e. $M[\kappa]$, our algorithm tracks the *fraction* of all slots available for that category, which is denoted as $C[\kappa]$. Finally, our algorithm tracks the last value written by a store to a given location a, which is denoted as $Val[a]$.

Definition 5 *The length of a minimal chain from category κ is feasible, written $\lambda_{min}(\kappa)$, iff one of the following holds:*

- $\kappa = 0 \wedge C[\kappa] \geq 1$
- $\kappa = 1 \wedge C[\kappa] \geq 3$
- $\kappa = 2 \wedge C[\kappa] \geq 4$
- $\kappa = 3 \wedge C[\kappa] \geq 5$

Definition 6 *Let $i, j, k \in \{1, 2, \cdots, p\}$ with $i \neq j, j \neq k, i \neq k$. Given a category κ, the breadth required for a minimal chain is feasible, written $\beta_{min}(\kappa)$, iff one of the following holds:*

- $\kappa = 0 \wedge av^i \geq 1$ *for some* i
- $\kappa = 1 \wedge av^i \geq 1 \wedge av^j \geq 2$ *for some* i, j
- $\kappa = 2 \wedge av^i \geq 3 \wedge av^j \geq 3$ *for some* i, j
- $\kappa = 3 \wedge av^i \geq 1 \wedge av^j \geq 3 \wedge av^k \geq 3$ *for some* i, j, k

Definition 7 *We say that a minimal chain from category κ is feasible, written $\phi_{min}(\kappa)$, iff $\lambda_{min}(\kappa) \wedge \beta_{min}(\kappa)$, i.e. when both its length and breadth are feasible.*

Definition 8 *Let $X = \{x \in \mathcal{Z}_+ \mid n/p - av^i\}$ be the set of slots from thread T^i which are filled with operations and let $X_{x>m} = \{x \in X \mid x > m\}$. The last slot of a thread T^i containing a reference to location a is:*

$$max_a^i = \begin{cases} 0 & \textit{if } X = \emptyset \vee \forall x \in X \ (T^i[x] = O_{b \neq a}) \\ m & \textit{if } T^i[m] = O_a \wedge \forall x \in X_{x>m} \ (T^i[x] = O_{b \neq a}) \end{cases}$$

Figure 3 describes the top-level routine of the proposed generator. That routine reserves n/p slots for the operations in each thread (line 4) and evaluates the overall number of slots available for operations belonging to chains of each category (line 6). Then the routine performs chain generation (lines 7-14). Chain generation builds as many chains as feasible by ramdomly selecting a category and a location for the conflicting endpoints of each chain. The building of a chain starts only if a minimal chain from the selected category is feasible (line 9), otherwise that category is excluded from the set of available ones (line 13).

```
 1: $\mathcal{A} \leftarrow \{a_1, a_2, \cdots, a_s\}$
 2: $\mathcal{K} \leftarrow \{0, 1, 2, 3\}$
 3: **for** $i \leftarrow 1, 2, \cdots, p$ **do**
 4:     $av^i \leftarrow n/p$
 5: **for** $\kappa \leftarrow 0, 1, 2, 3$ **do**
 6:     $C[\kappa] \leftarrow n \times M[\kappa]$
 7: **repeat**
 8:     $\kappa \leftarrow$ random $\mathcal{K}$
 9:     **if** $\phi_{min}(\kappa)$ **then**
10:         $a \leftarrow$ random $\mathcal{A}$
11:         chain $(\kappa, a)$
12:     **else**
13:         $\mathcal{K} \leftarrow \mathcal{K} - \{\kappa\}$
14: **until** $\mathcal{K} = \emptyset$
```

Figure 3: Algorithm Sequence-Generator(p, n, s, M)

Figures 4 and 5 describe the routines employed by the proposed generator. The pseudocode assumes that $\mathcal{K}$, $\mathcal{A}$, C, T^i and av^i all have global scope for those routines.

Figure 4 describes the routine that builds a chain from category κ whose endpoints are conflicting operations at location a. Essentially, the routine randomly selects locations, operations, processors, and lengths, unless otherwise constrained by the rules of formation of a given category. For categories 2 and 3, the routine starts building a minimal chain from the target category (lines 19-28 and 36-46), but before closing its construction, it decides whether a longer chain (from the same category) should be derived or not from the chain under construction (lines 29 and 47). If so, the chain is extended (lines 31 and 49); otherwise, the minimal chain is concluded (lines 32-34 and 50-51). Note that the precondition $\phi_{min}(\kappa)$ ensures that the sets in lines 2, 12, 13, 19, 20, 36, 37, and 38 are all non-empty. Three auxiliary routines are invoked: insert (for adding an operation to a chain), extend (for extending a minimal chain from categories 2 and 3), and sco (for imple-

```
if $\kappa = 0$ then
    $i \leftarrow$ random $\{x \in \{1,2,\cdots,p\} \mid av^x \geq 1\}$
    $\lambda \leftarrow$ random $\{1,2,\cdots,av^i\}$
    while $\lambda \geq 1 \wedge av \geq 1 \wedge C[\kappa] \geq 1$ do
        if $max_a^i \neq 0$ and $T[max_a^i]$ is $L$ then
            $op \leftarrow S$
        else
            $op \leftarrow$ random $\{L,S\}$
        insert$(\kappa, op_a^i)$
        $\lambda \leftarrow \lambda - 1$
if $\kappa = 1$ then
    $j \leftarrow$ random $\{x \in \{1,2,\cdots,p\} \mid av^x \geq 2\}$
    $i \leftarrow$ random $\{x \in \{1,2,\cdots,p\} \mid av^x \geq 1 \wedge x \neq j\}$
    insert$(\kappa, S_a^i)$
    insert$(\kappa, L_a^j)$
    $op \leftarrow$ random $\{L,S\}$
    insert$(\kappa, op_a^j)$
if $\kappa = 2$ then
    $i \leftarrow$ random $\{x \in \{1,2,\cdots,p\} \mid av^x \geq 3\}$
    $j \leftarrow$ random $\{x \in \{1,2,\cdots,p\} \mid av^x \geq 3 \wedge x \neq i\}$
    $op \leftarrow$ random $\{L,S\}$
    insert$(\kappa, op_a^i)$
    insert$(\kappa, MB^i)$
    $b \leftarrow$ random $\mathcal{A} - \{a\}$
    $op \leftarrow$ random $\{L,S\}$
    insert$(\kappa, op_b^i)$
    $op \leftarrow$ sco $(\kappa, op, b, i)$
    insert$(\kappa, op_b^j)$
    $\lambda \leftarrow$ random $\{0,3,6,\cdots,C[\kappa]\}$
    if $\lambda \neq 0$ then
        $j \leftarrow$ extend$(\kappa, a, j, \lambda)$
    insert$(\kappa, MB^j)$
    $op \leftarrow$ random $\{L,S\}$
    insert$(\kappa, op_a^j)$
if $\kappa = 3$ then
    $j \leftarrow$ random $\{x \in \{1,2,\cdots,p\} \mid av^x \geq 3\}$
    $i \leftarrow$ random $\{x \in \{1,2,\cdots,p\} \mid av^x \geq 3 \wedge x \neq j\}$
    $h \leftarrow$ random $\{x \in \{1,2,\cdots,p\} \mid av^x \geq 1 \wedge x \neq j \wedge x \neq i\}$
    insert$(\kappa, S_a^h)$
    insert$(\kappa, L_a^i)$
    insert$(\kappa, MB^i)$
    $b \leftarrow$ random $\mathcal{A} - \{a\}$
    $op \leftarrow$ random $\{L,S\}$
    insert$(\kappa, op_b^i)$
    $op \leftarrow$ sco $(\kappa, op, b, i)$
    insert$(\kappa, op_b^j)$
    $\lambda \leftarrow$ random $\{0,3,6,\cdots,C[\kappa]\}$
    if $\lambda \neq 0$ then
        $j \leftarrow$ extend$(\kappa, a, j, \lambda)$
    insert$(\kappa, MB^j)$
    insert$(\kappa, L_a^j)$
```

Figure 4: Routine chain(κ, a)

menting the fourth clause of Definition 3). Those routines are described in Figures 5a, 5b, and 5c, respectively.

```
1: $T^i[n/p - av^i + 1] \leftarrow op$
2: $av^i \leftarrow av^i - 1$
3: $C[\kappa] \leftarrow C[\kappa] - 1$
```

(a) insert(κ, op^i)

```
 1: while $\lambda \geq 3 \wedge av^j \geq 2 \wedge c[\kappa] \geq 3$ do
 2:     $P \leftarrow \{x \in \{1, 2, \cdots, p\} \mid av^x \geq 3 \wedge x \neq j\}$
 3:     if $P \neq \emptyset$ then
 4:         return $j$
 5:     else
 6:         insert($\kappa, MB^j$)
 7:         $c \leftarrow$ random $\mathcal{A} - \{a\}$
 8:         $op \leftarrow$ random $\{L, S\}$
 9:         insert($\kappa, op^j_c$)
10:         $m \leftarrow$ random $P$
11:         $op \leftarrow$ sco $(\kappa, op, c, j)$
12:         insert($\kappa, op^m_c$)
13:         $j \leftarrow m$
14:     $\lambda \leftarrow \lambda - 3$
15: return $j$
```

(b) extend(κ, a, j, λ)

```
1: if $op = L \wedge av^i < 1$ then
2:     $O \leftarrow S$
3: else
4:     $O \leftarrow$ random $\{L, S\}$
5: if $op = O = L$ then
6:     insert($\kappa, S^i_b$)
7: return $O$
```

(c) sco(κ, op, b, i)

Figure 5: Auxiliary routines of the generator

The asymptotic time complexity of the sequence generation algorithm depends on how the computations required by Definitions 7 and 8 are implemented. Note that their propositions must be updated each time an operation is inserted in some slot. We assume an optimal implementation for those computations: they can be initialized, updated, and evaluated in times $O(p)$, $O(1)$, and $O(1)$, respectively. Let us first analyze the complexity of the algorithms in Figures 4 and 5. Since any random selection takes $O(1)$ and the routine *insert* (Figure 5a) also takes $O(1)$, the routine *sco* (Figure 5c) takes $O(1)$. The complexities of the routines *extend* (Figure 5b) and *chain* (Figure 4) are dominated by the computation of the subset of processors required to accommodate a given chain category (line 2 in routine *extend*; lines 2, 12, 13, 19, 20, 36, 37, and 38 in routine *chain*). Such computation takes $O(p)$. Therefore, the computation performed by the loops in the routines *extend* and *chain* take, respectively, $O(p) \times O(n_i) = O(p\ n_i)$ and $O(p\ n_i) + O(p) = O(p\ n_i)$, where n_i denotes the number of operations generated by the i-th invocation of routine *chain*. Now, let us analyze the complexity of the top-level routine (Figure 3). Initialization (lines 1-6) takes $O(s + p)$. To analyze the main loop (lines 7-14), note that the random selection (line 8), the evaluation of the proposition (line 9), and the removal of a category (line 13) all take $O(1)$. Therefore, the loop computation is dominated by the routine *chain*, which is $O(p\ n_i)$. Let I be the number of iterations of that loop. The computation of the loop takes $\sum_{i=1}^{I} O(p\ n_i)$. Since in the end of the loop, the number of operations is bounded (at most n), the loop takes $O(p\ n)$. Thus, the overall worst-case running time complexity of the sequence generation algorithm is $O(s + p) + O(p\ n)$, i.e. $O(p\ n + s)$.

It should be noted that the worst-case complexity of a conventional sequence generator is $O(p + n)$.

4 ADDRESS ASSIGNMENT

This chapter describes how biasing constraints are exploited when converting locations into effective addresses. First, it illustrates the key ideas and formalizes the notions required to formulate the target problem. Finally, it describes the algorithm proposed to solve the constrained random address assignment problem.

4.1 KEY IDEAS AND MAIN NOTIONS

4.1.1 Selection of a competition pattern

Let a cache *row* means either a cache block in a directed-mapped cache or a set in a set-associative cache. The main idea behind address assignment is the competition biasing constraint (*cbc*). For a given *cbc*, there are different ways in which addresses compete for the same cache row and they can be seen as patterns which can be cast into a graph representation, as follows:

Definition 9 *A competition pattern is an undirected graph $CP = (A, C)$ where each vertex $a_i \in A$ represents a location and each edge $(a_i, a_j) \in C$ represents the fact that a_i and a_j compete with each other for the same cache row.*

Figure 6 illustrates that notion for a verification scenario with four locations and a cache with 2^I rows.

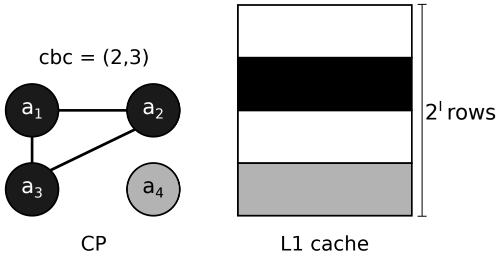


Figure 6: The interpretation of a competition pattern

For the sake of address assignment, a pattern should be randomly selected from a uniform distribution over all competition patterns induced by a given *cbc*. Figure 7 enumerates all competition patterns induced for $s = 4$. Since relation C (“compete for a cache row”) is reflexive, symmetric, and transitive, C is an equivalent relation and, therefore, it induces a partition $\{A_x\}$ of the set A such that each equivalent class A_x is represented by a strongly connected component. Besides, C is such that every strongly connected component turns out to be a clique. As a result, CP is a trivial form of a perfect graph and can, therefore, be characterized by two numbers: its clique cover number κ and its chromatic number χ. Therefore, we can define a *cbc* as a pair (κ, χ) with $\kappa, \chi \in \{1, 2, \cdots, s\}$.

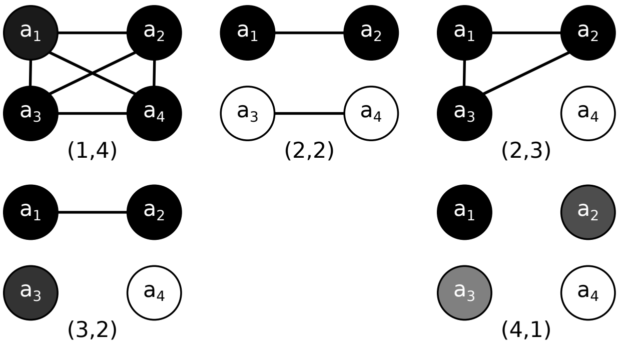


Figure 7: Enumeration of all CPs for $s = 4$

It should be noted that, for the cbcs $(2, 2)$, $(2, 3)$, and $(3, 2)$, there are isomorphic graphs which represent the same pattern. Such graphs correspond to distinct labelings of the vertices and are not shown in Figure 7. The adoption of a single pattern to represent all isomorphic graphs does not limit the verification space because a labeling is the result of a late phase of address assignment: the constrained random choice of *effective* addresses. (We show a preview of the actual labeling here for illustrative convenience only). However, a same *cbc* may induce a collection of non-isomorphic competition patterns. For instance, Figure 8 shows that, for $s = 8$, there are exactly two CPs induced by $cbc = (3, 4)$. Thus, to avoid limiting the verification space, the address assigner must be able to randomly select among them with probability $1/2$. Finally, notice that not all pairs of (κ, χ) represent feasible constraints. Fortunately, due to the simple topology, the collection of all feasible pairs for a given s can be easily precomputed with grounds on graph theory.

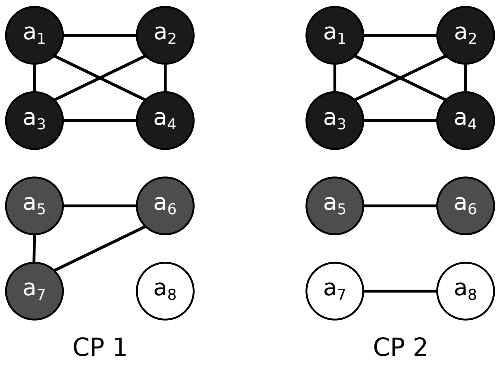


Figure 8: CPs induced by $cbc = (3, 4)$ for $s = 8$

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

For simplicity, when we refer to a *component* from now on, we mean a strongly connected component of a competition pattern.

4.1.2 Enforcement of biasing constraints

Given an *abc*, an *sbc*, and a competition pattern randomly selected from a *cbc*, they are enforced when assigning binary patterns to distinct address fields. Assume that an N-bit effective address consists of three fields: a block *offset* field with O bits (meaning that 2^O is the number of bytes in a cache block), an index field with I bits (meaning that the cache has 2^I rows), and a *tag* field with $T = N - O - I$ bits.

Figure 9 shows an example of how the effective addresses are enforced to comply with biasing constraints. The example corresponds to $abc = 2$ (word alignment), $sbc = true$ (true sharing), and assumes the pattern illustrated in Figure 6, which was induced by $cbc = (2, 3)$. To enforce such competition pattern, the same binary pattern must be assigned to the index field for vertices from the same clique (as indicated in black) and distinct binary patterns must be assigned to that field for vertices from different cliques (as indicated in gray). To enforce true sharing, distinct binary patterns must be assigned to the tag field for vertices from the same clique (as depicted in blue, red, and yellow), because this ensures that they will not lie in the same block (since their memory block addresses are different). Finally, to enforce word alignment, the two least significant bits from the offset field are set to zero. (Note that fields in white remain unconstrained).

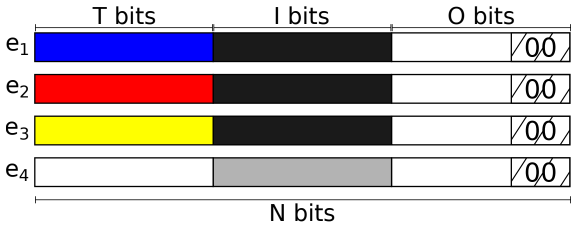


Figure 9: Relation between address fields and biasing constraints

4.2 PROBLEM FORMULATION

The general address assignment problem can be formulated as follows. Let E be the effective address space. Given an *abc*, an *sbc*, a feasible *cbc*, and a set of locations $A = \{a_1, a_2, \cdots, a_s\}$, find a mapping $\alpha : A \mapsto E$ such that all biasing constraints are satisfied.

This dissertation targets an instance of that problem, where competition patterns and binary patterns are randomly selected under uniform distributions. To formulate the target instance, we rely on the notions defined in the previous section, which can be encapsulated with the following notation.

Let $\mathcal{CP}_{\kappa,\chi,s}$ denote the collection of competition patterns induced by a given $cbc = (\kappa, \chi)$ and by a given number of shared addresses s. Let random $\mathcal{CP}_{\kappa,\chi,s}$ denote the random selection of a pattern from that collection.

Given a pre-selected pattern $CP = (A, C)$, let the collection $\{A_x\}$ denote the partition of the set A induced by its (strongly connected) components. Given an N-bit effective address, say e, let $e.\mathcal{O}$, $e.\mathcal{I}$, and $e.\mathcal{T}$ denote, respectively, its offset, index, and tag fields with O, I, and $T = N - O - I$ bits. Finally, let random $[0, 2^F - 1]$ denote the random selection of a binary pattern representing a number in the range $[0, 2^F - 1]$ for an address field with F bits.

The proposed technique solves the following random address assignment problem instance:

Problem 2 *Given the set of addresses E, the set of locations $A = \{a_1, a_2, \cdots, a_s\}$, an abc, and sbc, and a feasible cbc $= (\kappa, \chi)$, find an injective mapping $\alpha : A \mapsto E$ such that all the following conditions hold:*

1. $(A, C) =$ *random* $\mathcal{CP}_{\kappa,\chi,s}$
2. $\forall a_i \in A$: $\alpha(a_i).\mathcal{T} =$ *random* $[0, 2^T - 1]$
3. $\forall a_i \in A$: $\alpha(a_i).\mathcal{I} =$ *random* $[0, 2^I - 1]$
4. $\forall a_i \in A$: $\alpha(a_i).\mathcal{O} =$ *(random* $[0, 2^{O-abc} - 1]) \times 2^{abc}$
5. $a_i \in A_x \wedge a_j \in A_{u \neq x} \Rightarrow \alpha(a_i).\mathcal{I} \neq \alpha(a_j).\mathcal{I}$
6. $a_i, a_j \in A_x \Rightarrow \alpha(a_i).\mathcal{I} = \alpha(a_j).\mathcal{I}$
7. $a_i, a_j \in A_x \wedge sbc \Rightarrow \alpha(a_i).\mathcal{T} \neq \alpha(a_j).\mathcal{T}$

Note that Conditions 1 to 4 represent the random selection of competition and binary patterns from uniform distributions. Condition 4 also captures the *abc*. Conditions 5 and 6 enforce the *cbc*; Condition 7, the sbc.

The next section describes the proposed algorithm to solve Problem 2.

4.3 SOLVER ALGORITHM

To completely specify *one* among the (possibly) multiple competition patterns induced by a given *cbc* $= (\kappa, \chi)$, we employ a function f that defines how many components have a same number c of locations, i.e. $f(c)$ represents the number of components with cardinality c in a given competition pattern, as formalized below.

Definition 10 *Given a number s of locations and a feasible cbc $= (\kappa, \chi)$, an inducer for a competition pattern is a function $f\colon \{1, 2, ..., \chi\} \mapsto \{0, 1, ..., \kappa\}$ such that $s = \sum_{c \in \{1,2,...,\chi\}} c \times f(c)$, $\kappa = \sum_{c \in \{1,2,...,\chi\}} f(c)$, and $f(\chi) \geq 1$.*

Note that, since the collection of components forms a partition of the set of locations, each competition pattern can be specified by a unique inducer f. For this reason, our algorithms work on the inducer f, from which the corresponding competition pattern, written CP_f, can be built. Although distinct inducers are defined for distinct values of

κ, χ and s, for simplicity, we rely on the shorthand notation f (instead of $f_{\kappa,\chi,s}$), unless the meaning is not clear from the context.

To handle Condition 1 of Problem 2, we propose an algorithm that first enumerates all candidate patterns induced by a *cbc* $= (\kappa, \chi)$ for a given number of locations s, before selecting one among them randomly. The algorithm relies on two observations for iteratively enumerating all candidate patterns:

1. From any $\mathcal{CP}_{\kappa,\chi,s-1}$, a pattern can be induced for $\mathcal{CP}_{\kappa,\chi,s}$ by including an extra vertex in some pre-existing component (so as to preserve κ), but not in a component with cardinality χ (so as to preserve χ).

2. The base for enumeration should be a primal competition pattern constructed with the minimum number of locations, i.e. $\chi + \kappa - 1$ (one clique with χ vertices and $\kappa - 1$ cliques with a single vertex), as formalized below:

Definition 11 *Given a cbc* $= (\kappa, \chi)$*, the inducer for the primal competition pattern is:*

$$f^*(c) = \begin{cases} 1 & \textit{if } c = \chi \\ \kappa - 1 & \textit{if } c = 1 \\ 0 & \textit{otherwise} \end{cases}$$

Figure 10 describes the algorithm for selecting a competition pattern from a uniform distribution. Line 1 sets the current number of locations σ to the minimum. Line 2 creates an initial collection containing only the primal competition pattern. Lines 3-12 iterate over an increasing number of locations until the target number s is reached. Line 4 increments the current number of locations and line 5 properly initializes the new collection of competition patterns to be constructed in a given iteration. Lines 6-12 iterate over all competition patterns obtained for $\sigma - 1$ locations. For the inducer of a given competition pattern, lines 7-12 iterate over the cardinalities of its components, except for the one corresponding to the maximum clique ($c \neq \chi$). In every iteration, a new inducer is created (line 9), initially identical to the inducer of the current competition pattern. Then the new inducer is updated to reflect the inclusion of a vertex in one of the (possibly) multiple components with cardinality c. This is accomplished by decrementing the number of components with cardinality c (line 10) and by incrementing those with cardinality $c + 1$ (line 11). Next, the updated inducer is used to build the new competition pattern that is included in the collection under construction (line 12). When the target number of locations is reached (line 12), such collection contains all the required competition patterns. Finally, one of them is randomly selected with probability density of $1/|\mathcal{CP}_{\kappa,\chi,s}|$ (line 13).

```
1: $\sigma \leftarrow \chi + \kappa - 1$
2: $\mathcal{CP}_{\kappa,\chi,\sigma} \leftarrow \{CP_{f^*}\}$
3: while $\sigma < s$ do
4:     $\sigma \leftarrow \sigma + 1$
5:     $\mathcal{CP}_{\kappa,\chi,\sigma} \leftarrow \emptyset$
6:     for each $CP_f \in \mathcal{CP}_{\kappa,\chi,\sigma-1}$ do
7:         for each $c \in [1, \chi - 1]$ do
8:             if $f(c) \neq 0$ then
9:                 let $f'$ be a new inducer s.t. $f' = f$
10:                $f'(c) \leftarrow f(c) - 1$
11:                $f'(c+1) \leftarrow f(c+1) + 1$
12:                $\mathcal{CP}_{\kappa,\chi,\sigma} \leftarrow \mathcal{CP}_{\kappa,\chi,\sigma} \cup \{CP_{f'}\}$
13: return random $\mathcal{CP}_{\kappa,\chi,s}$
```

Figure 10: Routine select-competition-pattern($cbc = (\kappa, \chi)$, s)

Let us now explain how the intended mapping is built from the selected competition pattern. The key idea to enforce a feasible mapping by construction is the *iterative* pruning of the *available* address space after an index is assigned to a component of a competition pattern and after a location is mapped to an effective address. Let $\mathcal{E}$ denote the *available address space* at a given iteration. Assume that, while iterating over the locations forming a component, location a is mapped to address e. To enforce injection, $\mathcal{E}$ must be reduced to $\mathcal{E} \setminus \{e\}$ for the next iteration, since this precludes the reuse of e for future mappings. Assume that, while iterating over the components of a competition pattern, the index i is assigned to component A_x. Let $\mathcal{E}_i$ denote the set of available addresses induced by index i, i.e. $\mathcal{E}_i = \{\epsilon \in \mathcal{E} : \epsilon.\mathcal{I} = i\}$. To enforce the uniqueness of index i across distinct components (Problem 2, Condition 5), $\mathcal{E}$ must be reduced to $\mathcal{E} \setminus \mathcal{E}_i$ for the next iteration, because this precludes the use of any address with the same index for future components. Finally, assume that, while iterating over the locations of a component, location a is assigned to address e with index i and tag t, but true sharing is required. In this case, each tag must be unique within the scope of a component. Let $\mathcal{E}_{i,t}$ denote the set of available addresses induced by index i and tag t, i.e. $\mathcal{E}_{i,t} = \{\epsilon \in \mathcal{E} : \epsilon.\mathcal{I} = i \ \wedge \ \epsilon.\mathcal{T} = t\}$. To enforce tag uniqueness across the locations of a given component (Problem 2, Condition 7), $\mathcal{E}$

must be reduced to $\mathcal{E} \setminus \mathcal{E}_{i,t}$ for the next iteration, because this precludes the reuse of a tag t for mapping future locations of the same component, but not for locations forming other components. (Note that, since $e \in \mathcal{E}_{i,t}$, the reduction ends up by also enforcing injection in this case).

It should be noted that not all indices are candidates for assignment to a given component A_x, because the relation between locations and addresses must be a function. To ensure that no location is left unmapped (while enforcing injection iteratively), a candidate index i must satisfy one of the following conditions:

- When true sharing is not enforced, the available address space must contain at least as many addresses with same index i as the number of locations in A_x;
- When true sharing is enforced, the available address space must contain at least as many addresses with same index i and distinct tags as the number of locations in A_x.

This notion is formalized as follows. Let $\mathcal{E}_i^{\tau} \subset \mathcal{E}_i$ denote the subspace of the available addresses induced by the same index i and different tags, i.e.

$$\mathcal{E}_i^{\tau} = \{\epsilon \in \mathcal{E}_i : \forall \epsilon' \in \mathcal{E}_i \setminus \{\epsilon\} \;\; \epsilon.\tau \neq \epsilon'.\tau\}$$

Definition 12 *Given a component A_x, an sbc, and the available address space $\mathcal{E}$, the set of candidate indices, written $\mathcal{I}(A_x, sbc, \mathcal{E})$, is:*

$$\mathcal{I}(A_x, sbc, \mathcal{E}) = \begin{cases} \{i \in [0, 2^I - 1] : |\mathcal{E}_i| \leq |A_x|\} & \textit{if } \neg sbc \\ \{i \in [0, 2^I - 1] : |\mathcal{E}_i^{\tau}| \leq |A_x|\} & \textit{if } sbc \end{cases}$$

Figure 11 relies on the notions formalized above to describe the top-level routine of the proposed address assigner. Line 1 selects a competition pattern. Line 2 prunes from the address space the addresses not satisfying the alignment constraint and initializes the *available* address space. Lines 3-15 iterate over each component of the selected competition pattern. Line 4 randomly selects an index for the current component from the set of candidate indices. Line 5 builds the set of available addresses induced by the selected index, while line 15 reduces $\mathcal{E}$ to enforce index uniqueness across distinct components. Lines 6-14 iterate over each location a of the current component. Line 7 randomly selects an effective address e with the index selected for the current component. Line 10 builds the set of available addresses induced by the selected index and the assigned tag. Line 11 reduces $\mathcal{E}$ to enforce tag uniqueness within the same component (under true sharing). Both line 11 and line 13 remove the selected address from $\mathcal{E}$ to enforce injection. Line 14 maps a location to an effective address. Finally, line 16 returns the injective mapping satisfying all constraints of Problem 2.

```
 1: $CP \leftarrow$ select-competition-pattern($cbc$, $s$)
 2: $\mathcal{E} \leftarrow \{ \epsilon \in E : \epsilon.\mathcal{O} \in [0, 2^{O-abc} - 1] \times 2^{abc} \}$
 3: for each $A_x$ in $CP$ do
 4:     $i \leftarrow$ random $\mathcal{I}(A_x, sbc, \mathcal{E})$
 5:     $\mathcal{E}_i \leftarrow \{\epsilon \in \mathcal{E} : \epsilon.\mathcal{I} = i\}$
 6:     for each $a \in A_x$ do
 7:         $e \leftarrow$ random $\mathcal{E}_i$
 8:         $t \leftarrow e.\mathcal{T}$
 9:         if $sbc$ then
10:             $\mathcal{E}_{i,t} \leftarrow \{\epsilon \in \mathcal{E} : \epsilon.\mathcal{I} = i \wedge \epsilon.\mathcal{T} = t\}$
11:             $\mathcal{E} \leftarrow \mathcal{E} \setminus \mathcal{E}_{i,t}$
12:         else
13:             $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
14:         $\alpha(a) \leftarrow e$
15:     $\mathcal{E} \leftarrow \mathcal{E} \setminus \mathcal{E}_i$
16: return $\alpha$
```

Figure 11: Algorithm Address-Assigner(abc, cbc, sbc, s, E)

Let us first analyze the asymptotic time complexity of the algorithm that selects a competition pattern (Figure 10). Since $|\mathcal{CP}_{\kappa,\chi,\sigma-1}| \leq |\mathcal{CP}_{\kappa,\chi,\sigma}| \leq |\mathcal{CP}_{\kappa,\chi,s}|$ by induction, we can use $|\mathcal{CP}_{\kappa,\chi,s}|$ as an upper bound for $|\mathcal{CP}_{\kappa,\chi,\sigma-1}|$ in line 6 and for $|\mathcal{CP}_{\kappa,\chi,\sigma}|$ in line 12. Lines 2 and 9 take $O(\chi)$ to build a data structure that allows the computation of lines 8, 10, and 11 in time $O(1)$. The computations in all other lines also take $O(1)$, except for line 12, which takes $O(log\ |\mathcal{CP}_{\kappa,\chi,s}|)$. Let us build the complexity from the innermost to the outermost loop. The innermost loop (lines 7-12) is repeated at most χ times and its computation is dominated by lines 9 and 12. Therefore, it takes $O(\chi \times (\chi + log\ |\mathcal{CP}_{\kappa,\chi,s}|))$. The intermediate loop (lines 6-12) is repeated at most $|\mathcal{CP}_{\kappa,\chi,s}|$ times. Therefore, it takes $O(|\mathcal{CP}_{\kappa,\chi,s}| \times \chi \times (\chi + log\ |\mathcal{CP}_{\kappa,\chi,s}|))$. Finally, the outermost loop (lines 3-12) is repeated at most s times. Therefore, it takes $O(s \times |\mathcal{CP}_{\kappa,\chi,s}| \times \chi \times (\chi + log\ |\mathcal{CP}_{\kappa,\chi,s}|))$. Since the routine is dominated by line 2 and by the outermost loop, it takes $O(\chi + s \times |\mathcal{CP}_{\kappa,\chi,s}| \times \chi \times (\chi + log\ |\mathcal{CP}_{\kappa,\chi,s}|))$. For a given cbc (since κ and χ are constants), the worst-case complexity of the algorithm in Figure 10 is $O(s \times |\mathcal{CP}_{\kappa,\chi,s}| \times log|\mathcal{CP}_{\kappa,\chi,s}|)$.

Now, let us analyze the asymptotic time complexity of the address assignment algorithm (Figure 11). When the set of effective addresses (E) is a contiguous address subspace, the computations in all lines take $O(1)$, except for line (line 1). Since the inner loop iterates over each address of a component, the outer loop iterates over each component, and all components form a partition, the lines 3-15 have the joint effect of visiting each location exactly once. Therefore, the computations in lines 2-15 take $O(s)$, which is dominated by the complexity of line 1. Thus, the worst-case complexity of the algorithm in Figure 11 is the same as the algorithm in Figure 10, i.e. $O(s \times |\mathcal{CP}_{\kappa,\chi,s}| \times log|\mathcal{CP}_{\kappa,\chi,s}|)$.

However, if the set of effective addresses (E) is a non-contiguous address subspace, the computations in lines 2, 4, 5, 7, 10, 11, 13, and 15 all take $O(|E|)$, because they have to check if an effective address belongs to the set E. Therefore, the computations in lines 2-15 take $O(|E| + s \times |E|) = O(s \times |E|)$. Thus, in such condition, the worst-case complexity of the algorithm in Figure 11 becomes $O(s \times |\mathcal{CP}_{\kappa,\chi,s}| \times log|\mathcal{CP}_{\kappa,\chi,s}| + s \times |E|)$.

5 INSTRUCTION SYNTHESIS

This chapter describes the instruction synthesizer, which converts a sequence of operations into a sequence of instructions for the target architecture, but also enforces an important property required by memory-model checkers, as formalized next.

5.1 PROBLEM FORMULATION

Problem 3 *Given a collection* $\{T^i\}$ *of operation sequences and a mapping* $\alpha : A \mapsto E$*, find a collection of instruction sequences by replacing each location* $a \in A$ *by an effective address* $\alpha(a) \in E$ *such that* $Val[S_j] \neq Val[S_m] \neq Val^0_a$ *for every* $S_j, S_m \in \mathcal{S}_a$.

Note that the value constraint specifies that all the values written by stores conflicting at a given address must be unique. It should be noted that a few random test generators over-constrain all stores to assign unique values, regardless of address (e.g. (HANGAL et al., 2004)). Unlike them, our constraint represents a necessary and sufficient condition for identifying the store producing the value that is observed by a *conflicting* load, which is the actual property required by most memory-model checkers.

5.2 SOLVER ALGORITHM

Figure 12 describes the proposed algorithm to solve Problem 3. Instruction generation translates operations to actual instructions and ensures that unique values are assigned to each conflicting store (line 12).

In our implementation, the functions invoked in lines 6, 10, and 13 actually generate C code, which is then compiled to the target architecture so as to obtain the native instructions[1].

Since the generation of each instruction takes constant time (regardless of the parameters), the worst-case running time of the algorithm is $O(n+p)$.

[1]To preserve the intended order of instructions defined by the generator, all compiler optimizations that could reorder instructions were disabled.

```
1:  for each $a \in A$ do
2:      $Val[a] \leftarrow 0$
3:  for $i = 1, 2, \cdots, p$ do
4:      for $x = 1, 2, \cdots, n/p - av^i$ do
5:          if $T^i[x]$ is membar then
6:              generate_M($T^i[x]$)
7:          else
8:              let $op_a = T^i[x]$
9:              if op = L then
10:                 generate_L($\alpha(a)$)
11:             else
12:                 $Val[a] \leftarrow Val[a] + 1$
13:                 generate_S($\alpha(a), Val[a]$)
```

Figure 12: Algorithm Instruction-synthesizer($\{T^i\}, \alpha$)

6 EXPERIMENTAL EVALUATION

This chapter compares a generator built with the proposed techniques with a conventional random test generator. It first describes the conditions of the experiments and defines the metrics adopted for comparison. Then it evaluates both generators according to each metric.

6.1 EXPERIMENTAL SET UP

To obtain representations for the designs under verification, we relied on the gem5 simulator infrastructure (BINKERT et al., 2011). We selected the out-of-order CPU timing model (*O3*), the *system call emulation* mode, the *Ruby* model for the memory subsystem, and the *simple* model for the interconnect network. We adopted the SPARC instruction set architecture and Alpha's (GHARACHORLOO, 1995) memory (consistency) model as verification target. Designs were derived from 8, 16, and 32-core architectures. The selected microarchitectures consisted of private L0 (split) caches, private L1 (unified) caches, shared L2 cache, and a coherence engine relying on a three-level MESI invalidation-based directory protocol[1]. L0, L1, and L2 correspond to 4KB (directed-mapped), 64KB (2-way), and 2MB (8-way) caches, respectively, all operating with the same block size (64 bytes) and the same replacement policy (LRU).

Starting from a correct design for a given architecture, we built 5 derivative designs containing each a single, *different* type of design error. Table 1 describes the errors in terms of the state machines of the cache controllers in the memory hierarchy. Design errors were built by modifying the original states machines, either by causing a transition to a wrong state or by precluding some output action associated with a transition. For reproducibility, we employ in Table 1 the same labels used in the gem5's infrastructure (available from https://www.m5sim.org/). Note that, as errors were injected one at time to derive distinct designs, we can precisely determine whether a given error was exposed or not by a test program, thereby ruling out (for

[1]It corresponds to changeset 10224 from the revision available at https://repo.gem5.org/gem5/rev/54d3ef2009a2. We are aware that such protocol version contains a few bugs reported in the literature (ELVER; NAGARAJAN, 2016). To cope with them, a test program is applied to a derivative design containing one of our artificial errors only after ensuring that it does not detect a bug on the original design representation.

evaluation purposes) the interference that would exist with multiple errors in a single design.

Table 1: Description of classes of artificial design errors

ID	Location	Current state	Input event	Next state	Precluded output action
F1	L1 cache controller	E_IL0	L0_DataAck	MM	u_writeDataFromL0Response
F2	L1 cache controller	M_IL0	WriteBack	MM_IL0	u_writeDataFromL0Request
F3	L0 cache controller	E	Store	E instead of M	–
F4	L1 cache controller	IS	Data_Exclusive	E	u_writeDataFromL2Response
f29	L1 cache controller	M_IL0	L0_DataAck	EE instead of MM	–

We selected the collection of design errors in Table 1 (from a broader set) according to the following criteria: 1) errors should be chosen by analyzing their behavior from the perspective of a conventional generator; 2) all errors should be exposed by a conventional generator for at least one combination of parameters within the ranges defined for the experiment; 3) each error should exhibit a quite different behavior such that it could serve as an archetype for similar errors.

Two characteristics were exploited to distinguish error types: how the probability of detection behaves with the growing number of cores and how it behaves with the growing number of operations sharing the same location, which we call the *sharing level* of a test program. Let us informally summarize our choice according to the third criterion, as follows. F1 is easier to expose under intensive sharing, largely regardless of core up-scaling. F2 can be exposed regardless of sharing level and core up-scaling. F3 becomes harder to expose with core up-scaling, regardless of sharing level. F4 becomes easier to expose with core up-scaling, largely regardless of sharing level. f29 is easier to expose with core up-scaling but only under intensive sharing. Appendix A provides a more precise characterization for such errors.

To evaluate the impact of the proposed technique, we compared the following pseudorandom generators according to three distinct metrics (potential for error exposure, actual effectiveness, and resulting effort). PLAIN+ is a typical generator, which is similar to the ones used for memory-model checking (e.g. (HANGAL et al., 2004; SHACHAM et al., 2008)). Since we could not find one available in the public domain, we relied on the pseudocode reported in (RAMBO; HENSCHEL; SANTOS, 2011) to implement our own prototype. However, for a fair comparison, we replaced the primitive address assigner used in that generator by the one proposed in Chapter 4. CHAIN+ is a generator exploiting both mechanisms proposed in this dissertation: the sequence generator from Chapter 3 and the address assigner from Chapter 4.

All generators have *common program parameters* that enforce general properties of the test to be generated: number of threads (p), number of memory operations (n), and number of shared locations (s). All generators have a common parameter for pseudorandom generation (seed) and common parameters for address biasing constraints. However, a generator has specific parameters tied to its inner mechanism for sequence generation. PLAIN+ relies on *instruction* mixes specifying the target proportions of load, store, and membars (whose values were inspired by related works (RAMBO; HENSCHEL; SANTOS, 2012)). CHAIN+ relies on *category* mixes specifying target proportions of chain categories (whose values were obtained empirically)[2]. Table 2 shows the target mixes.

Table 2: Target mixes

Instruction mix			Category mix			
Load	Store	Membar	$\mathcal{C}_0$	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_3$
0.30	0.66	0.04	0.40	0.60	0.00	0.00
0.48	0.48	0.04	0.00	1.00	0.00	0.00
0.66	0.30	0.04	0.00	0.80	0.20	0.00
0.80	0.16	0.04	0.00	0.80	0.00	0.20

To verify each derivative design of a given architecture, distinct test suites were synthesized with each generator. We compared the generators for a same setting of the common program parameters by letting the others vary within pre-defined ranges and by defining a metric on the collection of tests they induce. We call a *verification scenario* the collection of all random tests induced by a same setting of parameters (p, n, s) when distinct mixes and different seeds are explored.

To select ranges for the common program parameters, we relied on values reported from industrial verification environments (ADIR et al., 2004; MANOVIT; HANGAL, 2006; HU et al., 2012). Tests for post-silicon usage contain hundreds of thousands of operations (MANOVIT; HANGAL, 2006; HU et al., 2012) and a few hundreds of shared locations (MANOVIT; HANGAL, 2006). Tests for pre-silicon usage contain tens of thousands of operations (ADIR et al., 2004). Therefore, since intensive data races are key to error exposure, the number of shared locations should be kept in the order of a few tens for reaching the same level of inter-processor conflict required by the best post-silicon practices. How-

[2]When randomly selecting an operation for a chain, the generator employs the following probabilities: 0.75 for loads and 0.25 for stores.

ever, for a wider evaluation of the impact of the proposed technique on verification effectiveness, we adopted a range that also includes values one order of magnitude smaller.

Each generator synthesized tests exploring the same parameters: 5 program sizes (n = 4000, 8000, 16000, 32000, 64000), 4 amounts of shared locations (s = 4, 8, 16, 32), 15 distinct random seeds (1, 2, 3, $\cdots$ 15), and 4 different target mixes. As a result, the test suites generated by PLAIN+ and CHAIN+ contain 1200 programs that are applied to each of the 5 derivative designs containing errors (i.e. 6000 use cases per architecture).

When evaluating the impact of the proposed sequence generator, each verification scenario was constrained by a *single cbc*, the same for both PLAIN+ and CHAIN+. For the verification scenarios with s = 4, 8, 16, 32, the following *cbc* values were employed, respectively: (4,1), (7, 2), (13,4), (15,8)[3]. For every generator, all addresses were aligned to the block ($abc = 2^6$) and true sharing was enforced ($sbc = true$).

We say that a test program *exposes* a design error if it leads the checker to detect a *violation* of the memory model specification. Among the two pre-silicon runtime checkers reported in the literature, we adopted the one providing full verification guarantees (FREITAS; RAMBO; SANTOS, 2013), so as to avoid that false negative or false positive diagnoses could underestimate or overestimate error exposure.

Runtimes were measured in an HP xw8600 Workstation (based on Intel Xeon E5430, 2.66 GHz) with an 8GB main memory.

6.2 METRICS

6.2.1 Error exposure correlation

To evaluate whether or not both generators were able to expose an error for a *same* setting of their (common) parameters, we enumerated all the combinations of parameter values within the adopted ranges and we quantified the impact of parameter choice on the joint potential of the generators for error exposure. To do so, we first quantified the potential exposure and then we performed the correlation, as follows.

[3]The *cbc* values were selected so that exactly 25% of all shared locations compete for the same cache row, while each of the 75% remaining non-competing locations are mapped to a distinct cache row.

To quantify the *potential for error exposure*, we adopted the following procedure for each error. For a given verification scenario (p, n, s), we generated multiple random tests by exploring distinct seeds and mixes. Then we applied the multiple tests to a design containing a given error. If *at least one* test led to the detection of that error, we marked the verification scenario as "exposing". Next, we repeated that procedure for the same error in all verification scenarios.

To perform the correlation, we adopted the following procedure for each error. For each verification scenario (p, n, s), we checked whether or not both generators have marked such scenario as "exposing" for a given error. If so, we labeled (p, n, s) as a scenario of *joint exposure* (written CHAIN+.PLAIN+). If not, we labeled it as a scenario of *mutually-exclusive exposure*, depending on whether it was an "exposing" scenario for PLAIN+ only (written PLAIN+.not CHAIN+) or for CHAIN+ only (written CHAIN+.not PLAIN+). Otherwise, it was labeled as a *joint non-exposure* scenario (written not CHAIN+.not PLAIN+). Finally, we computed the percentage of all verification scenarios with distinct labels for a given error. Note that those under joint exposure can be interpreted as the collection of parameter settings for which the generators are correlated with respect to potential exposure.

6.2.2 Effectiveness

To comparatively evaluate the *effectiveness* of the generators in exposing a *given error* in a *given verification scenario*, we measured the *fraction* of all test programs (induced by that scenario) for which violations were detected. This fraction could be interpreted as the probability of a generator to expose that error (assuming sufficient sampling). Then we calculated the average effectiveness of a given error on a collection of verification scenarios and, finally, on the collection of all errors by using the simple arithmetic mean.

6.2.3 Verification effort

To estimate the overall *verification effort* of running the random tests induced by a given verification scenario in an *attempt* to expose an error, we combine the effectiveness and the average test runtime measured for the error in that scenario. The deliberately simple example in Figure 13 illustrates their relation. Since the effectiveness estimates the probability of a test to expose an error, its inverse represents the probable number of random tests required to expose that error (on average). Therefore, the average time required to expose an error is the ratio between average test runtime and effectiveness. For a given verification scenario, large effectiveness is translated into small effort, because a small number of random tests (all with the same setting of parameters) is required to expose the error. However, when the effectiveness is zero for a given verification scenario, none of the induced random tests was able to expose the error. In this case, the (wasted) effort corresponds to the runtime for fully executing all random tests induced by that scenario. These notions are formalized in the following.

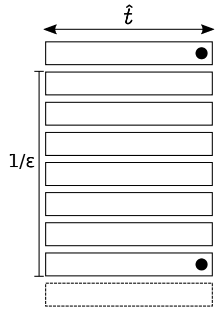


Figure 13: The relation between effectiveness (ε), average test runtime ($\hat{t}$) and effort ($\hat{t}/\varepsilon$). The bars represent the many tests from a suite, the length of a bar represents the (average) runtime of a test, and dots denote the hypothetical instants when an error would be successively exposed if simulation was not interrupted.

Let $T = \{T_i\}$ denote the collection of tests induced by a given verification scenario and let t_i denote the measured runtime for test T_i. Let ε denote the effectiveness measured for the error in that scenario. Let $\hat{t} = \sum_{T_i \in T} t_i/|T|$ denote the average test runtime over all tests in that scenario. Given an error and the collection of random tests (synthesized with generator G) for a given verification scenario, we estimate the average effort spent when those tests try to expose that error as follows:

$$EF_G = \begin{cases} \hat{t}/\varepsilon & \text{if } \varepsilon \neq 0 \\ |T|.\hat{t} & \text{if } \varepsilon = 0 \end{cases}$$

Note that, when an error is bound to be undetectable under a given setting of parameters, an increase in the number of generated random tests will only increase the wasted effort.

Therefore, to compare the relative improvement in the effort required to *actually expose* an error, the evaluation must be constrained to the verification scenarios where both generators expose that error (joint exposure). In such scenarios, we evaluate the relative effort as follows:

$$\frac{EF_{PLAIN}}{EF_{CHAIN}} = \frac{\hat{t}_{PLAIN}}{\hat{t}_{CHAIN}} \times \frac{\varepsilon_{CHAIN}}{\varepsilon_{PLAIN}}$$

After defining the effort for a given error and a given verification scenario, we calculated the average effort of a given error over a collection of verification scenarios and, finally, over the collection of all errors by using the simple arithmetic mean.

6.3 IMPACT OF THE PROPOSED SEQUENCE GENERATION

This section reports two evaluation approaches. First, it provides a broad assessment by exploring the full ranges adopted for the parameters (the goal is to determine how to properly set them for maximizing the error exposure obtained with each generator). Then it focuses on sub-ranges that better reflect the practical use of each generator (the goal is to compare the generators when each is operating at its best).

For the broad assessment, we analyzed the results for the whole set of verification scenarios. Note that such set comprises different families of potential test suites for distinct target architectures (different values for p). Notice also the *wide* parameter ranges: for instance, the number of operations was varied within one order of magnitude. Even for a fixed core count, it is unlikely that a real-life test suite may contain tests obtained with such a wide variation of parameters. To help mimic the pragmatic use of the generators, an in-depth assessment should focus on more meaningful subsets of scenarios. By analyzing the verification outcomes over the full parameter ranges, we found out that the generators required quite distinct numbers of shared locations to reach best exposure[4]. That is why we compared them when each generator exploited the location count leading to its best exposure. The criterion adopted to define *best exposure* was the following: we partitioned the verification scenarios as a collection where each set was induced by a distinct value of s; then, among the sets exposing the most errors, we selected the one with the maximum number of tests exposing errors.

[4]The reasons for that will be explained in the following subsections.

This corresponds to $s = 8$ for PLAIN+ and $s = 32$ for CHAIN+.[5]

The next subsections compare the generators in terms of distinct metrics and reflect the two evaluation approaches described above.

6.3.1 Impact of parameter choice on error exposure

Table 3 shows, for each error, the percentages of verification scenarios with potential for error exposure. The table has three partitions: one reports joint exposure, another reports mutually-exclusive exposure, yet another reports the overall exposure induced by each generator.

Table 3: Percentage of verification scenarios with potential for error exposure

Exposed by	F1	F2	F3	F4	f29	avg
PLAIN+ . CHAIN+	2%	80%	68%	3%	5%	32%
PLAIN+ . not CHAIN+	8%	0%	3%	7%	2%	4%
CHAIN+ . not PLAIN+	32%	12%	7%	7%	3%	12%
PLAIN+	10%	80%	71%	10%	7%	36%
CHAIN+	34%	92%	75%	10%	8%	44%

Joint exposure for a same setting of parameters is evidence that error detection is more likely determined by the setting itself than by the specific features of each generator. Therefore, on average, PLAIN+ and CHAIN+ seem indistinguishable in terms of error exposure for 32% of all the verification scenarios. (However, in those scenarios, they may differ in the effort required to uncover errors. This is why Section 6.3.3 will compare their relative effort for joint exposure scenarios).

On the other hand, mutually-exclusive exposure for a same setting of parameters is evidence that error detection is more likely determined by the specific features of one of the generators than by proper parameter setting. Therefore, those are the scenarios that actually distinguish the generators in terms of error exposure. In such scenarios, CHAIN+ was superior for four errors (F1, F2, F3, f29). On average, CHAIN+ exposed errors not exposed by PLAIN+ in 12% of all verification scenarios while the opposite held for 4% of them. This means that CHAIN+ was superior in extending the number of parameter set-

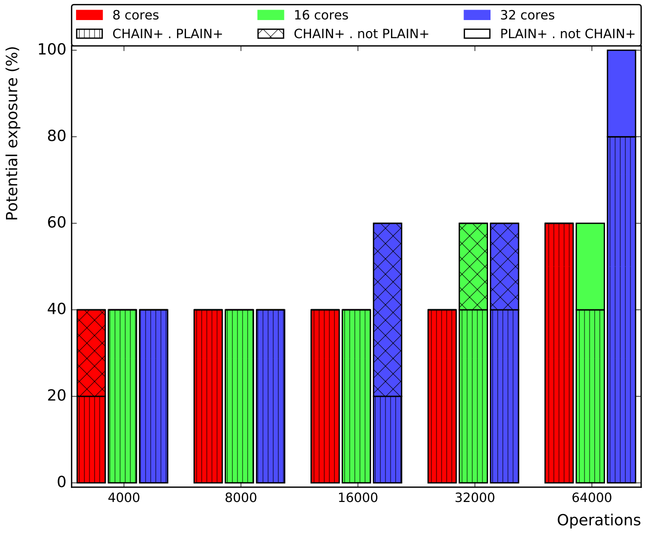
[5]This holds for all but 8-core architectures, where PLAIN+ is at its best exposure with $s = 16$ instead of $s = 8$. Since both values lead to quite similar exposure, for simplicity, let us consider $s = 8$ as a unified value for PLAIN+'s best exposure over all architectures.

tings leading to error detection, i.e. CHAIN+ makes error exposure less sensitive to parameter choice. If compared with respect to joint exposure, CHAIN+ extends the number of verification scenarios leading to detection by 38% (12/32) on average, while PLAIN+ extends it by 13% (4/32). In particular, albeit CHAIN+ extended error exposure up to 4 times (32/8) for error F1 as compared to PLAIN+, the latter did not exhibit a similar improvement for any error.

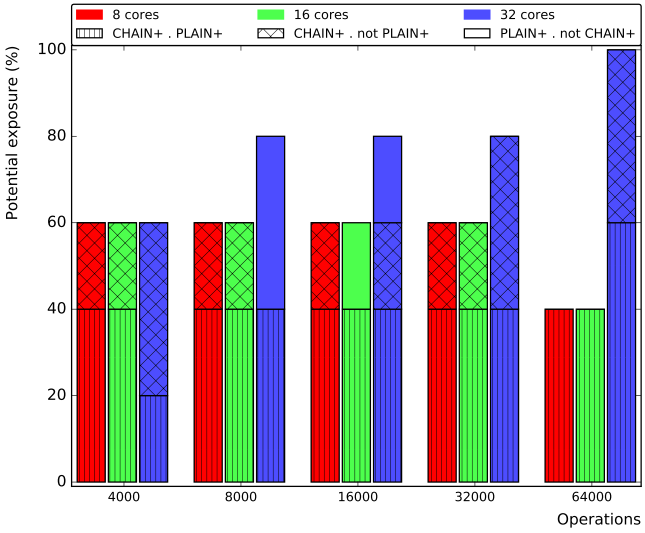
The third partition of Table 3 captures both mechanisms leading to exposure: proper choice of parameters and adequate generation features. On average, CHAIN+ and PLAIN+ exposed errors in 44% and 36% of all verification scenarios, respectively, i.e. CHAIN+ improved exposure by 22%. (It should be noted that each percentage in that partition represents an upper bound on effectiveness, which will be precisely determined in Section 6.3.2). Appendix B provides supplementary comparisons for a few subsets of the verification scenarios.

To illustrate how exposure behaves as a function of test length, we constrained the number of shared locations (s=8 and s=32 only) and computed the percentage of errors exposed in each verification scenario by one or both generators. Figure 14 shows that percentage as a function of test length under distinct numbers of locations. It provides a breakdown of the total percentage in terms of joint and mutually-exclusive exposure. In scenarios with the fewest locations (Figure 14a), at most 40% and 60% of all errors were exposed when test length is limited to 8000 and 32000 operations, respectively, and CHAIN+ always led to equal or more exposure as compared to PLAIN+. For the scenario with 64000 operations, albeit higher joint exposure could be obtained (in two scenarios), PLAIN+ was prominent in extending the exposure for the highest core counts. For the largest number of locations (Figure 14b), CHAIN+ was the prominent generator in extending exposure. However, when comparing Figures 14a with Figures 14b, the exposure was reduced in the scenario with 64000 operations for the lowest core counts.

To explain the distinct behavior of the generators with the number of locations, let us analyze in more detail the extreme scenarios in Figure 14 for the largest number of cores. Figure 15 shows a breakdown of the percentages of verification scenarios leading or not to detection for 32-core architectures (averaged on the whole collection of errors) when restricting the numbers of shared locations (s=8 and s=32) and the number of operations (n=4000 and n=64000). For the shortest tests, CHAIN+ and PLAIN+ exhibited the same capability of exposure with the smallest number of locations, but CHAIN+ led to three



(a) 8 shared locations



(b) 32 shared locations

Figure 14: Exposure as a function of test length

times more exposure for the largest number of locations. On the other hand, for the longest tests, PLAIN+ exposed errors in all the verification scenarios with the smallest number of locations, while CHAIN+ exposed errors in all verification scenarios with the highest number of locations. This can be explained as follows.

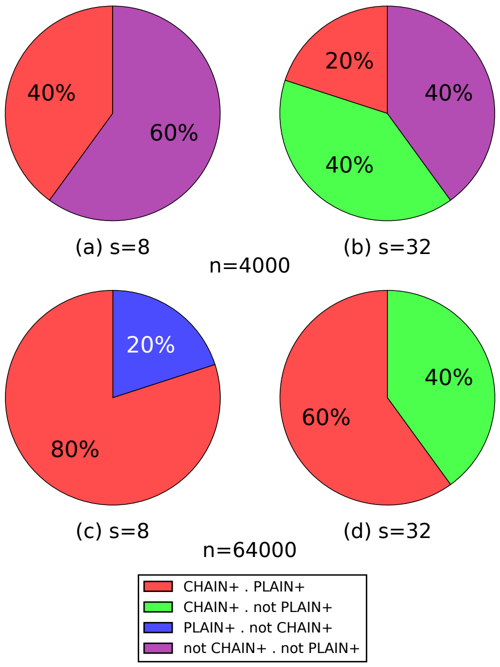


Figure 15: A breakdown for error exposure capability (p=32)

On the one hand, PLAIN+ independently selects a location for a memory operation without any correlation with the locations chosen for other operations. Therefore, the probability of creating test programs with intensive sharing is raised by adjusting the number of locations s to be small, because location choices are uncorrelated.

On the other hand, CHAIN+ does not independently select a location for a memory operation. Such location is often determined by the location chosen for other operations (in the same chain). Therefore, the probability of creating test programs with intensive sharing is not raised by simply minimizing the parameter s, because location choices are correlated. For instance, successive operations in a chain must share the same location if they lie in distinct processors, which is the key mechanism exploited by CHAIN+ to expose errors. However, the location selected for one endpoint of a chain *cannot be reused* within the chain, but only at the other endpoint. That is why a higher number of locations is required to reach the same level of sharing as a conventional generator while maintaining the advantages of chains as redundancy reductors.

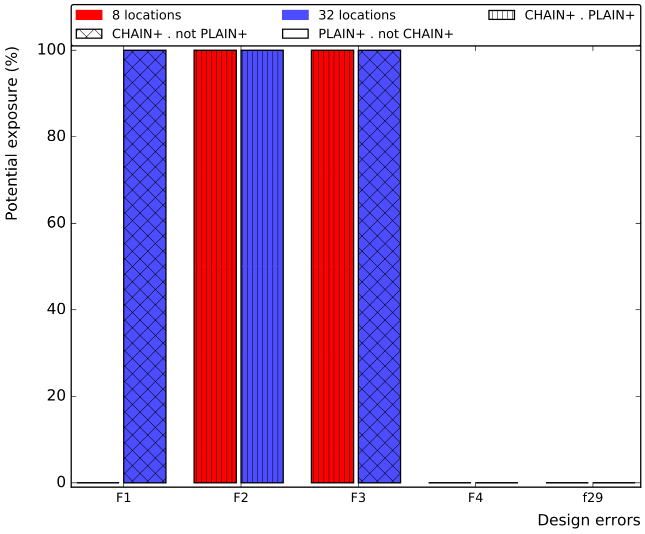
For those reasons, CHAIN+ *appears* to increase error exposure with less intensive sharing. Such apparent paradox comes from the conventional random test generation rationale, which is based on the assumption of uncorrelated selection. Besides, the higher number of locations required by CHAIN+ is not a price to pay for its advantages, because average test runtime indeed decreases with the number of shared locations (as it will be shown in Section 6.3.3).

Figure 16 shows the potential of each generator in exposing every error for a 32-core architecture when restricting the number of shared locations to 8 or 32 and the test length to 4000 or 64000 operations. (It should be noted that the main information conveyed by this figure lies less on bar height, but more on whether a bar is present or not and on what pattern it contains). For the shortest tests (Figure 16a), only errors F2 and F3 were exposed by every generator in both verification scenarios. CHAIN+ was the only generator able to expose error F1 (when the highest number of locations was exploited). No generator was ever able to expose errors F4 and f29. For the longest tests (Figure 16b), all errors were exposed by every generator in both verification scenarios, except that CHAIN+ did not expose F1 with the smallest number of locations. By contrasting Figures 16a and 16b, we found a first piece of evidence that CHAIN+ can expose more errors with shorter tests.

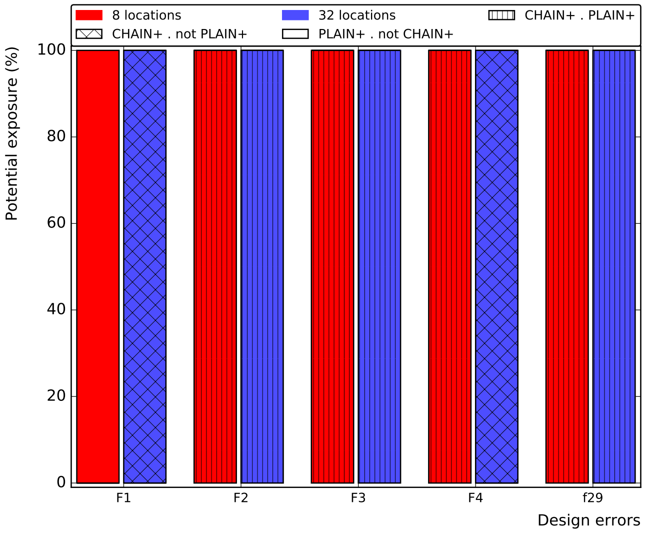
Remind, however, that 100% of exposure does not mean that all tests exposed each error (because each scenario consists of multiple tests induced by distinct seeds and mixes and not all of them may have led to detection). That value represents an upper bound for the actual effectiveness. The next subsection further refines the comparison between CHAIN+ and PLAIN+, by taking into account how many *tests* of every verification scenario actually led to error detection.

6.3.2 Impact on effectiveness

To report the improvement in effectiveness of CHAIN+ with respect to PLAIN+, we performed the following procedure on the set



(a) n=4000



(b) n=64000

Figure 16: Exposure per error for a 32-core architecture

of verification scenarios leading to joint exposure (i.e. PLAIN.CHAIN). First, for each generator, we measured the fraction of all tests induced by a given verification scenario that were able to detect a *same* error. Second, we calculated the relative effectiveness of CHAIN with respect to PLAIN by taking the ratio between the respective fractions. Then, we obtained such fractions for all verification scenarios leading to joint exposure. Next, we averaged the relative effectiveness for that error on that set, and determined its maximum and the minimum ratios (best and worst improvements).

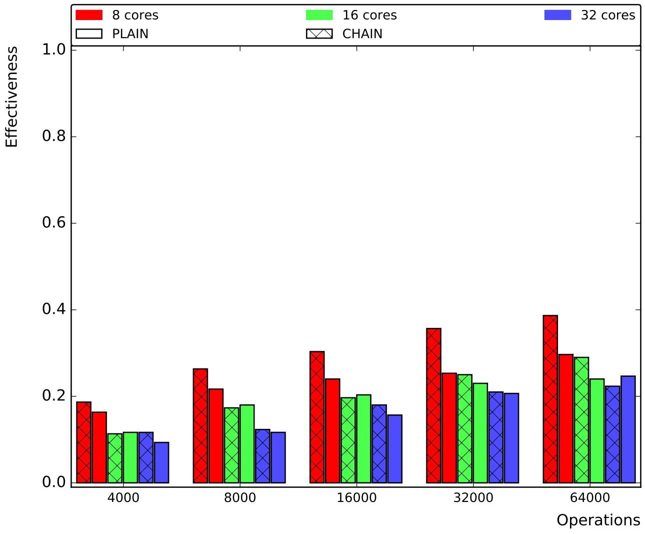
Table 4 reports the relative improvement ($\frac{\varepsilon_{CHAIN}}{\varepsilon_{PLAIN}}$) for each error in the best, worst, and average cases under verification scenarios leading to joint exposure. Note that, on average, CHAIN+ is more effective than PLAIN+ for three errors (F2, F3, and f29) and PLAIN+ is more effective than CHAIN+ for a single error (F4). Albeit PLAIN+ was twice as efficient as CHAIN+ in the worst case, CHAIN+ was up to 13 times more efficient than PLAIN+ in the best case.

Even though CHAIN+ and PLAIN+ were equally efficient for error F1 under joint exposure (Table 4), CHAIN+ led to the highest mutually-exclusive exposure for that error (Table 3). Albeit, on average, CHAIN+ is only marginally more effective for error F2 in joint-exposure scenarios (Table 4), CHAIN+ is the only generator able to expose F2 in 12% of all verification scenarios (Table 3). However, albeit PLAIN+ and CHAIN+ have similar potential to expose error F4 (Table 3), PLAIN+ is 50% more effective (Table 4). The reasons for that will be explained in Section 7.2. Supplementary comparisons for a few subsets of the verification scenarios are available in Appendix B.

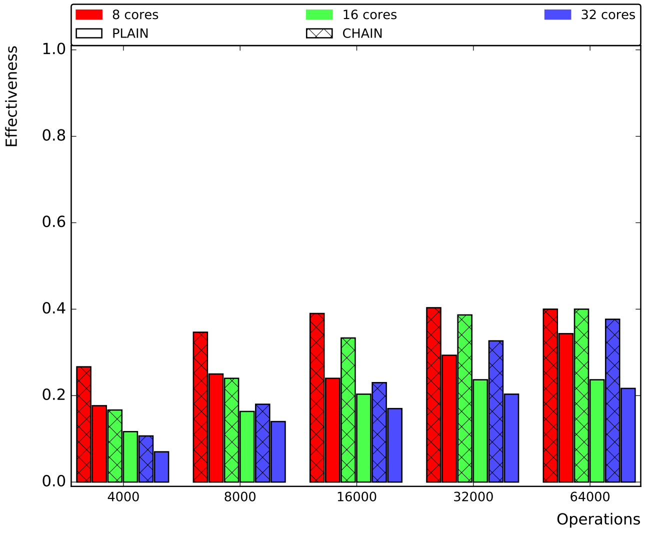
Table 4: Improvement in effectiveness under joint exposure

Errors	**Best**	**Worst**	**On average**	
F1	1.00	1.00	1.00	0%
F2	2.00	0.90	1.08	+**8%**
F3	13.00	0.50	3.92	+**292%**
F4	0.50	0.50	0.50	-50%
f29	2.00	1.00	1.33	+**33%**

Figure 17 shows, for each architecture, the effectiveness as a function of test length, calculated over the set of derivative designs with the implementation errors specified in Table 1 in all exposure scenarios (joint and mutually-exclusive). Note that, under $s = 32$, CHAIN+ was always more effective than PLAIN+ for all test lengths and every architecture. Besides, for a same architecture and a same test length,

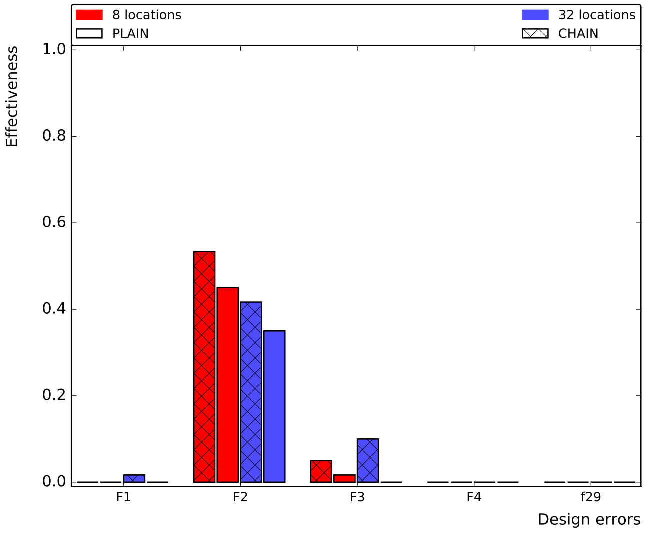


(a) 8 shared locations

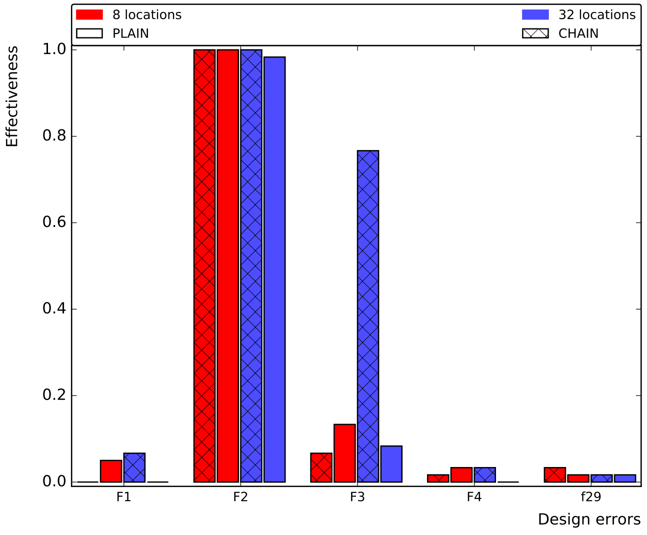


(b) 32 shared locations

Figure 17: Effectiveness as a function of test length



(a) n=4000



(b) n=64000

Figure 18: Effectiveness per error for a 32-core architecture

CHAIN+ was always more effective for $s = 32$ than PLAIN+ was for $s = 8$. This means that, albeit PLAIN+ seems more effective when exploiting few locations and CHAIN+ more effective when exploiting many locations, CHAIN+ is, on average, the most effective generator.

Figure 18 shows the effectiveness of each generator in exposing every error for a 32-core architecture when restricting the number of shared locations to 8 or 32 and the test length to 4000 or 64000 operations.

For the shortest tests (Figure 18a), both generators exposed F2 and F3 (but PLAIN+ detected F3 in a single scenario). CHAIN+ was also able to expose F1 when the highest number of locations was used.

For the longest tests (Figure 18b), on the other hand, all errors were exposed by both generators. PLAIN+ detected all of them for the smallest number of locations while CHAIN+ detected them all for the largest number. On the other hand, for the largest number of locations, PLAIN+ detected all but F1 and F4; for the smallest number, CHAIN+ detected all but F1. Finally, CHAIN+ led to the highest effectiveness for all errors but F4 and f29.

6.3.3 Impact on effort

This section first addresses effort under joint exposure (so as to distinguish the generators in verification scenarios where their exposure is indistinguishable). Then it evaluates the effort over all exposure scenarios (so as to compare their overall effort over a wide set of parameters). Finally, it focuses on best exposure scenarios (so as to better reflect the pragmatic use of the generators).

Table 5 reports the relative improvement ($\frac{EF_{PLAIN}}{EF_{CHAIN}}$) for each error in the best, worst, and average cases under verification scenarios leading to joint exposure.

Note that, on average, as compared to PLAIN+, CHAIN+ requires less effort for three errors (F2, F3 and f29) and more effort for two errors (F1 and F4). This means that, in general, CHAIN+ may require less effort most often than PLAIN+ and still lead to error detection. However, albeit CHAIN+ requires 6% more effort when both generators expose F1 (Table 5), CHAIN+ exposes it, in general, more often than PLAIN+ (Table 3). This means that, in general, CHAIN+ is likely to require less effort for 32% of all verification scenarios while PLAIN+ is likely to require less effort for only 8% of them (Table 3). On the other hand, for error F4, PLAIN+ requires around 50% less effort as compared to CHAIN+ for equivalent verification scenarios. Since both generators have the same mutually-exclusive exposure for F4 (Table 3), PLAIN+ is certainly the generator requiring less effort for exposing that error when *all* exposure scenarios are considered. Indeed, this points out a current limitation of the experimental setup that offers an opportunity for future improvement. That is why this error deserves a more detailed analysis to reveal the mechanisms preventing CHAIN+ to uncover it more often under joint exposure. Such analysis is postponed to Section 7.2.

Table 5: Improvement in effort under joint exposure

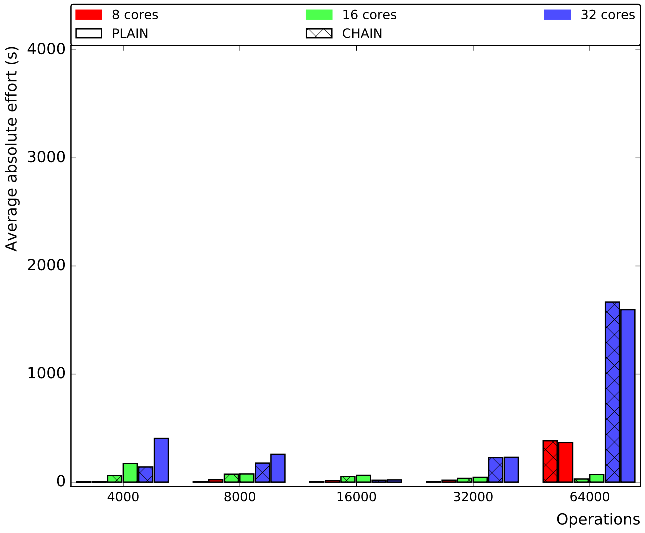
Errors	**Best**	**Worst**	**On average**	
F1	0.94	0.94	0.94	-6%
F2	2.02	0.78	1.05	+**5%**
F3	14.87	0.50	5.23	+**423%**
F4	0.53	0.51	0.52	-48%
f29	2.07	0.96	1.35	+**35%**

Let us now quantify the effort as a function of test length. Figures 19 and 20 contrast the effort's behavior for joint-exposure (CHAIN.PLAIN) and overall-exposure scenarios (CHAIN.PLAIN, CHAIN.not PLAIN, PLAIN.not CHAIN).

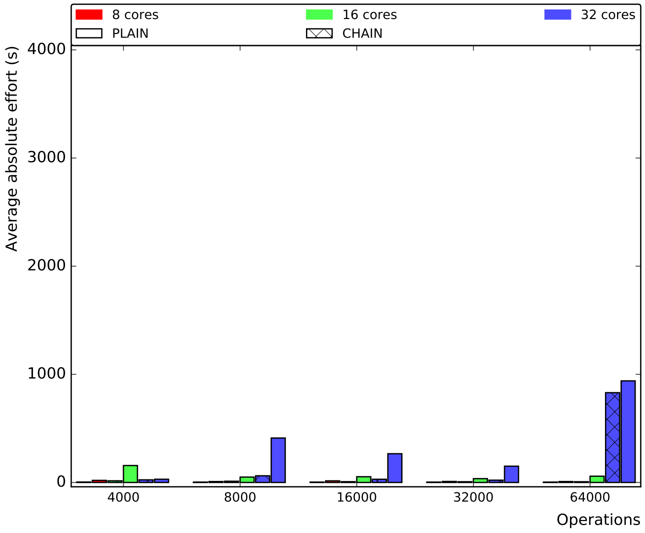
Figure 19 shows that, as compared to PLAIN+ under joint exposure, CHAIN+ only required significantly more effort for long tests operating over few locations (Figure 19a). For tests handling many locations (Figure 19b), CHAIN+ always required less effort regardless of test length.

Albeit Figure 20 exhibits a similar behavior, it shows a general amplification of effort under overall exposure. This illustrates how sensitive the effort is to parameter setting. When mutually-exclusive scenarios come into play, the average effort is amplified because many more scenarios happen to reflect a poor choice of parameters for one of the generators, especially for tests over few locations (Figure 20a). Note that, for tests over many locations (Figure 20b), despite the general amplification, CHAIN+ is essentially the one requiring less effort regardless of test length.

Besides, note that the effort is reduced from Figure 20a to Figure 20b for every verification scenario and each generator. Thus, if CHAIN+ is used to generate tests over a large number of locations, it is likely to require less effort than PLAIN+ on average, regardless of number of locations and test length.

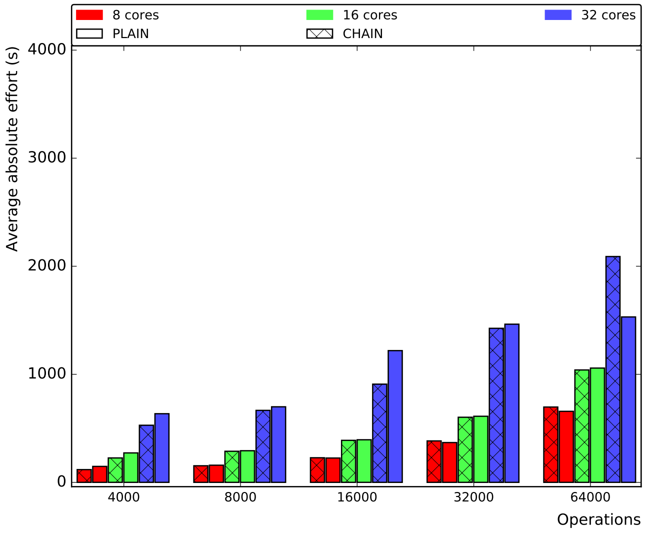


(a) 8 shared locations

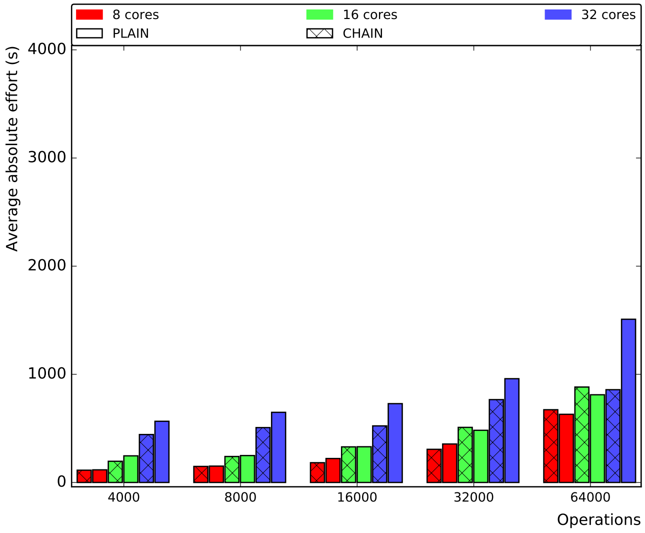


(b) 32 shared locations

Figure 19: Effort as a function of test length (joint exposure)



(a) 8 shared locations



(b) 32 shared locations

Figure 20: Effort as a function of test length (overall exposure)

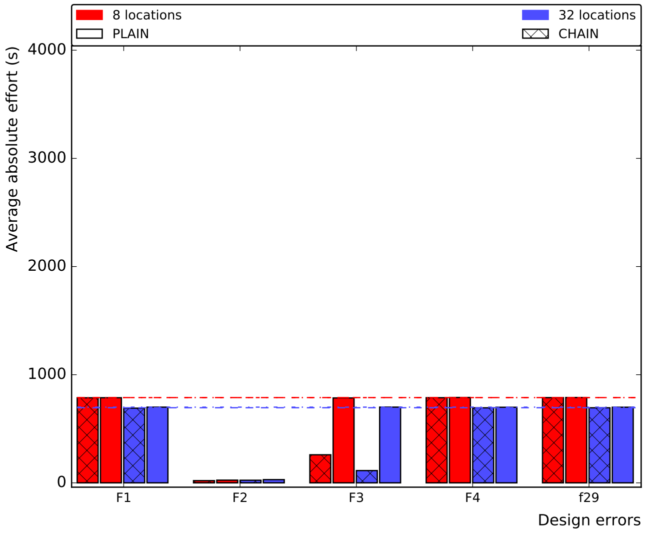
Finally, by contrasting Figures 19 and 20, we conclude that the scenarios inducing joint exposure happen to be the most advantageous for reducing the effort. Thus, those scenarios can hint the range of parameters for which CHAIN+ is more likely to minimize the required effort. Fortunately, the results in Section 6.3.1 can be exploited for this purpose. Figure 14b shows that joint exposure is dominant for test lengths with 16000 operations or more. This is a piece of evidence that CHAIN+ is likely to be *effective and efficient* for tests with tens of thousands of operations, exactly the range considered adequate for pre-silicon verification in industrial environments (ADIR et al., 2004).

It should be noted that Figures 19 and 20 try to mimic the average effort of having multiple errors in a design. However, the effort is highly sensitive to error type. To illustrate that, Figure 21 reports, for a 32-core architecure, the effort required when *trying* to expose each error in extreme verification scenarios. The dotted lines in the figure represent the total runtime required for fully executing all tests synthesized by a generator for a given verification scenario (60 random tests in this case). Since tests synthesized with the same common parameters by distinct generators may lead to different test runtimes, two dotted lines appear for each verification scenario. This means that, when a bar touches a dotted line, the error was not detected and the effort was wasted[6]. However, when an error is detected, the effort essentially reflects its inverse proportionality with the effectiveness of each generator.

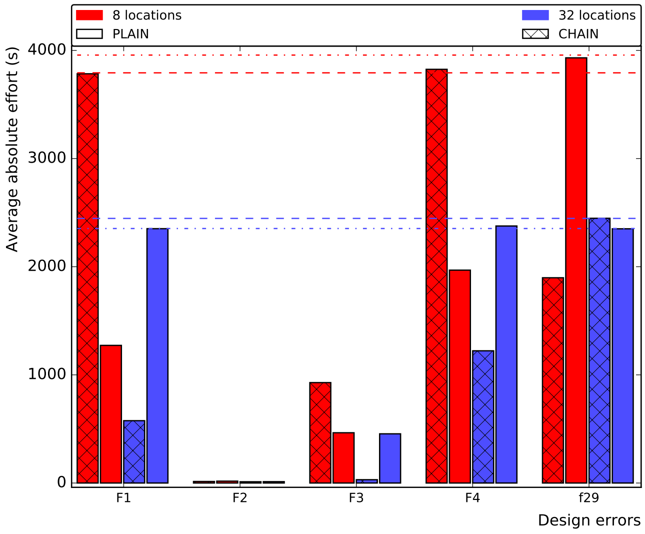
For short tests (Figure 21a), the effort is smaller, but most of it is wasted, because few errors were detected (no generator ever exposed F4 nor f29, PLAIN+ never exposed F1 nor F3, CHAIN+ did not exposed F1 and F3 in half of the verification scenarios). As a result of poor detection in such many cases and of joint detection in a single case (F2), both generators led to quite similar effort, except for the lower effort required by CHAIN+ to expose F3.

For long tests (Figure 21b), the higher effort was the price to pay for the detection of all errors, but not in all verification scenarios (in half of them, PLAIN+ or CHAIN+ did not expose F1 and PLAIN+ did not expose F4). Note that, when it generates tests exploring many locations, CHAIN+ required less or essentially the same effort as PLAIN+ to expose all errors. The same was not always true for PLAIN+ when exploring few locations.

[6]Indeed, it may also mean that, after running $|T|-1$ tests, the error was hit in the very end of the $|T|$-th test.



(a) n=4000



(b) n=64000

Figure 21: Effort per error for a 32-core architecture

Let us now focus on scenarios in which each generator is operating at its best exposure. Table 6 shows the improvement in effort when both generators expose each error (with distinct parameters). Therefore, such table captures the improvement in *useful* effort only. Note that, once operating at its best exposure, on average, CHAIN+ becomes more efficient than PLAIN+ for all errors (albeit the latter is also operating at its best exposure). To understand the causes for the reduction of useful effort, let us contrast Table 5 with Table 6. Both compare useful effort, but the former captures the effort when both generators rely on the same parameters; the latter, on distinct parameters. Therefore, the superiority of CHAIN+ comes less from the higher number of random tests exposing errors in a *given* verification scenario, but more from the higher number of verification scenarios whose tests are likely to expose errors. After evaluating the useful effort, let us compare the effort spent whether or not a test happens to expose an error. Table 7 shows the overall effort regardless of detection, i.e. it includes the contribution of *wasted* effort. Note that CHAIN+ is still more efficient than PLAIN+ (on average) for all derivative designs, even when a few errors might not have been detected, say, in the end of a test cycle. Such reduction in effort in one cycle would save time for the next test cycle.

Table 6: Improvement in useful effort under best exposure (both generators expose every error but with distinct parameters)

Errors	**Best**	**Worst**	**On average**	
F1	2.21	2.21	2.21	+**121%**
F2	1.85	0.87	1.14	+**14%**
F3	16.87	4.45	10.76	+**976%**
F4	1.61	1.61	1.61	+**61%**
f29	1.61	1.61	1.61	+**61%**

Still under best exposure, let us now focus on the most challenging verification scenario for each generator: the use of tests with 64000 operations to check 32-core designs (which corresponds to Figure 21b when PLAIN+ and CHAIN+ exploit, respectively, 8 and 32 locations). In such conditions, both generators were able to expose all types of errors, but CHAIN+ required from 1.5 to 15 times less effort than PLAIN+ to expose each error, except for F4 (for which a degradation of 19% was observed), for reasons to be explained in Section 7.2.

Effort reductions favor test throughput, which is an important factor in pseudorandom testing, because the more test cycles run, the

Table 7: Improvement in overall effort under best exposure (one or both generators may not expose some error)

Errors	**Best**	**Worst**	**On average**	
F1	3.01	0.96	1.45	+**45%**
F2	1.85	0.87	1.14	+**14%**
F3	34.47	4.45	13.88	+**1288%**
F4	3.01	0.98	1.27	+**27%**
f29	1.61	0.97	1.17	+**17%**

higher the confidence in the design (MANOVIT; HANGAL, 2006). Albeit the reduction in effort helps improving test throughput, it is the generator's effectiveness that serves as confidence holder. Therefore, a generator requiring shorter tests to expose more errors seems the most suitable for sustaining proper test throughput.

Table 8: Test length required to expose each error in 32-core designs (under best exposure)

Test Length (Operations)	Errors exposed by									
	CHAIN+					PLAIN+				
	F1	F2	F3	F4	f29	F1	F2	F3	F4	f29
4000	✓	✓	✓				✓	✓		
8000		✓	✓				✓	✓		
16000	✓	✓	✓				✓			
32000	✓	✓	✓	✓			✓	✓		
64000	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

That is why Table 8 shows how many errors each generator could expose for a given test length when targeting 32-core designs. Note that, as compared to PLAIN+, CHAIN+ exposed two types of errors as often (F2 and f29), one type of error 11% more often (F3), one type of error twice as often (F4), and one type of error 4 times as often (F1). Besides, the tests generated by CHAIN+ seem more inclusive with respect to test length (all errors exposed by a test were also exposed by a longer test, except in one case).[7]

It should be noted that the average test runtime decreases with the growing number of locations (i.e. in Figure 21, the red dotted lines are above the blue dotted lines), and this effect is especially prominent

[7]The fact that a couple of errors (F1 and F3) were not alwalys detected for larger programs (albeit detected for shorter ones) is due to the probabilistic nature of the generators.

for the longest tests. This can be explained as follows. For a given test length, the higher the number of locations, the smaller the probability of operation conflict. Under a relaxed memory model, non-conflicting operations issued by the same processor do not contribute to lengthen the chain of dependent events modeling shared-memory behavior within an event-driven simulator. Therefore, the less likely the conflict, the smaller the time required by the simulator to execute that chain. That is why this effect is more prominent for tests with the longest threads. Note that this grants CHAIN+ an extra advantage. The larger number of locations not only increases its effectiveness and, therefore, reduces the *useful* effort required to expose errors, but it also reduces the *wasted* effort when a test happens not to uncover an error (possibly saving time for extra tests).

Finally, we measured the runtime required by both generators to synthesize tests targeting 32-core designs with 64000 operations (when CHAIN+ and PLAIN+ explored, respectively, 32 and 8 locations). On average, CHAIN+ and PLAIN+ spent 2.65 and 0.54 seconds, respectively. Despite being 5 times slower than PLAIN+, the generation effort required by CHAIN+ is still 2 to 3 orders of magnitude smaller than the verification effort required to expose all errors, except for F2, which is an error easy to find. (For this error, the smallest measured verification effort was 11.5 seconds, i.e. around 4 times the generation effort). Thus, the higher generation effort required by CHAIN+ is negligible in face of the much larger verification effort required to expose hard-to-find errors.

7 CONCLUSIONS AND PERSPECTIVES

This chapter first draws our conclusions, then discusses hurdles for error exposure, and finally addresses further work.

7.1 CONCLUDING REMARKS

As compared to real-life synchronized programs, random tests with races expose shared-memory errors faster (MANOVIT; HANGAL, 2006). There are two main reasons for that, depending on the type of inter-processor conflict inducing the race: 1) A conflict between two stores exposes the serialization of writes, the main requirement for coherence; 2) A conflict between a store and a load exposes what value is returned by the load (through a uniprocessor or multiprocessor dependence chain), a main requirement for memory consistency. For those reasons, this dissertation exploited canonical dependence chains to improve the effectiveness of random test generation and, therefore, to reduce the effort required to expose errors.

Since design errors are accidental and obviously do not result from design specifications, there are not such things as typical design errors. Besides, most shared-memory errors reported in the literature are not described in full detail so as to be properly reproduced for experimental evaluation. Therefore, the best we could do was to create quite different types of artificial errors to challenge the generators.

We compared the proposed generator with a conventional one for a collection of hard-to-expose design errors under a varied set of verification scenarios (when using the same proposed technique for the sake of address assignment). Based on such scope of evaluation, we concluded that, when a sufficiently large number of shared locations is preselected, the proposed generator requires shorter tests for exposing design errors as compared to a conventional generator. We found out that the proposed generator not only benefits from the smaller number of operations to reduce the effort required for error detection, but it also benefits from the fact that the choice of a larger number of locations also contributes to reduce test runtime.

7.2 DISCUSSION

This section elaborates on a current limitation of the experimental setup that prevented CHAIN+ to expose error F4 as often as PLAIN+ (as already pointed out in Section 6.3.3). Despite focusing on a specific error (for illustrative purposes), our ultimate goal is to discuss general mechanisms hampering error exposure.

The adopted MESI protocol maintains inclusion with the upper-level caches (this means that, if a block is in cache L0, then it must also be in cache L1). When cache L2 happens to respond with a block requested by a local processor and asserts its exclusivity, error F4 disrupts inclusion because it precludes proper block allocation in cache L1 (but not in cache L0). Since the transition to state E is not precluded (see Table 1), error F4 ends up validating the contents of a stale block, which will be taken for the contents of the actual block. Therefore, if a remote processor requests that block, when a message reaches the cache L1, the stale block will be returned (albeit the cache L0 might still have the correct block), because the L1 controller assumes inclusion.

Under a precondition to enforce exclusivity, there is a sequence[1] of events able to expose error F4, as follows:

- **Precondition**: A given location is not previously allocated in any cache (except perhaps the last-level cache).
- **Condition 1**: First, the local processor issues a load from the given location;
- **Condition 2**: Then no intermediate memory accesses happen to modify the block containing the location's copy in cache L0 (otherwise it might be written back to L1, hiding the error);
- **Condition 3**: Next, a remote processor issues a load from the same location.

To illustrate the difficulty of exposing error F4 with CHAIN+, Figure 22 reproduces the example in Figure 2f, which illustrates a chain from category 2, and shows a couple of slight variations of that chain.

[1] Indeed, an alternative sequence can be obtained by keeping the Precondition and Condition 1, but slightly changing Conditions 2 and 3, as follows: **Condition 2a**: Then some intermediate memory accesses happen to evict the (unmodified) block containing the location's copy in cache L0 (since the protocol maintains inclusion, that copy is not written back to L1); **Condition 3a**: Next, the local processor issues a load from the same location.

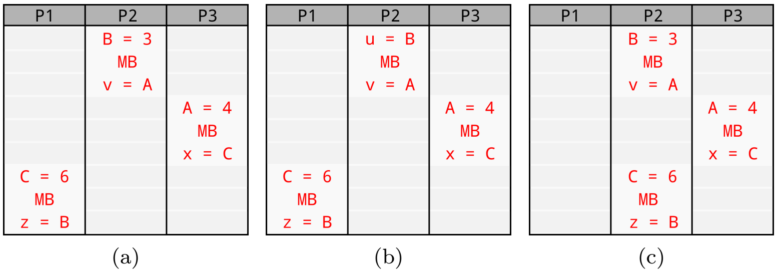


Figure 22: Candidate chains to expose an error

In Figure 22a, the chain illustrates that the rules of the category *guarantee* that Condition 2 holds by construction. Besides, that chain *happens* to satisfy Condition 3. Since the chain starts at the first operation of a thread, it also happens to satisfy the precondition. Unfortunately, since it starts with a store, the chain does not satisfy Condition 1. Therefore, the chain is unable to expose error F4.

Figure 22b shows a variant of that chain, which obeys the rules of the same category, but starts with a load. For this chain, the precondition and all conditions hold. Therefore, it is able to expose error F4.

Assume that, unfortunately, the generator decided for building the chain in Figure 22a. This means that the error will be exposed only if another chain is built and it happens to satisfy all conditions. Let us now consider how the first chain induces hurdles for error exposure. Note that, when the operations in the chain are executed, copies of locations A and C will be allocated in the private caches of distinct processors. Such allocations disqualify locations A and C as candidates for exposing the error, because they do not satisfy the precondition. This illustrates an inherent limitation of CHAIN for detecting the error: the different locations referenced between the chain endpoints, albeit automatically satisfying Condition 3, end up making the precondition false, *unless the respective blocks are evicted* from the private caches.

For instance, to restore the precondition for location A (making it a candidate for error exposure by another chain), evictions would be required to eliminate copies of that location in private caches. However, the sequence generator has no control on evictions, because they depend on the actual effective addresses assigned to the locations.

Let us illustrate the coupling between chain generation and address assignment by means of an example. Figure 22c illustrates an alternative chain with exactly the same locations as the chains discussed before. Assume 2-way (LRU) L0 caches and suppose that locations A, B, and C were assigned to effective addresses that compete for the same set. After the execution of the first two operations in the chain, copies of locations B and A will be allocated in the same set. When the store to location C is executed, the copy of location B will be evicted from P2's private cache. As a result, location B satisfies the precondition for the following load (which fulfills Condition 1). Albeit Conditions 3 and 4 must be fulfilled by later operations before the error can be exposed, this example illustrates how evictions allow a location to be reused by restoring the proper precondition for future chains.

This discussion has shown that the proposed approach is sound: although it decouples sequence generation from address assignment, the latter was conceived to accept (alternative) competition pattern constraints, which are keys to eviction control. This means that, the effort required by CHAIN to expose errors like F4 can be probably reduced with proper *cbc* selection. Albeit the approach was functionally validated, it was not fully evaluated yet. This is left as future work, as explained in the next section.

7.3 FUTURE WORK

To overcome a few current limitations of the proposed technique, we envisage conceptual generalizations, a technical extension, and extra experimentation.

7.3.1 Conceptual generalizations

A few simplifying assumptions were made to develop the algorithms. As a result, the following opportunities are open for improvement:

- **Relax restriction on category selection**: The current algorithm for sequence generation relies on mixes specifying the target proportions of chain categories. The ultimate goal of the adoption of chains for test generation is to avoid redundant sequences by exploiting the constraints embedded in the chain rules. However, the fixed choice of chain proportions limits the verification space, because it may end up synthesizing directed tests instead of random (albeit constrained) tests. That is why we intend to allow the generator to *randomly* select the target proportions of chain categories. Besides, we intend to replace the category mix with an instruction mix (similar to the one used by the plain generator) for specifying the target proportions of loads and stores (so as to constrain the random selection of operation type).

- **Relax implicit assumption on cache address space**: the current address assigner implicitly assumes one of the following scenarios: 1) the virtual and the physical address spaces coincide for the effective address subspace (unmapped addresses); 2) the cache is virtually addressed and aliasing is precluded by some mechanism (design limitations or operating system intervention). In such scenarios, the proposed address assigner offers full guarantees of properly constraining the verification space. However, when the cache is virtually indexed but physically tagged, albeit the **cbc** constraint is preserved (since the index is not translated), the constraint **sbc=true** may not be preserved through address translation. When the cache is physically indexed and physically tagged, neither **cbc** nor **sbc** can be guaranteedly preserved through translation. Fortunately, the **abc** constraint is guaranteedly preserved in all scenarios (since the least significant bits of the address are not translated).

- **Generalize the handling of competition patterns for lower hierarchical levels**: in the current address assigner, the handling of competition patterns was deliberately restricted to caches at the first level of hierarchy. We intend to generalize such handling towards the lower-level cache.

7.3.2 Technical extension

A few simplifying choices were made in the prototype, leading to the following opportunities for extension:

- **Couple address assigner and linker script**: The current prototype relies on ad hoc allocation of a range of addresses for representing the available address space. For the actual verification tool, the available address space should reflect the range of addresses defined by a linker script.

7.3.3 Extra experimentation

Despite the large number of use cases employed to obtain the reported results, there are many opportunities for further extending experimental evaluation:

- **Evaluate the impact of the new address assigner**: Although we proposed a new address assignment algorithm and we have shown that it works properly by using it as the address assigner module of both generators PLAIN+ and CHAIN+, we have not yet evaluated the impact of the proposed address assigner as compared to a conventional one. To do so, another version of the conventional random test generator should be created, say PLAIN, with a conventional address assignment module. We intend to compare PLAIN with PLAIN+ and PLAIN with CHAIN+ for a varied set of biasing constraints.

- **Evaluate the generators with respect to coverage**: As a metric for coverage, we intend to adopt the approach employed in (ELVER; NAGARAJAN, 2016), which estimates structural coverage by means of code coverage. To do so we intend to instrument the design representation so as to track the lines of code that are reached when each test is run. Then, we will measured the (cumulative) fraction of all lines reached when all tests of a given verification scenario are run. Finally, we will measure the (cumulative) fraction of all lines reached on the collection of all verification scenarios.

- **Evaluate the impact of contention in the network**: Albeit the required controllability to form the chains was assessed under a quite accurate timing model (the adopted design representation relied on gem5's O3 for CPU and Ruby for memory), we have relied on a simple model for the interconnect (gem5's *simple* network). To assess the impact of contention on the chain-based mechanism, it should be evaluated under a more elaborate network model (gem5's Garnet model). Albeit such evaluation might not be useful for pre-silicon verification, it could pave the way towards the porting of the chain-based generator for post-silicon testing.

- **Extend and characterize the collection of errors**: It is not as easy as it may seem to conceive an artificial design error that is subtle enough to actually challenge the generators. Although there has been continuous work-in-progress to increase the number of errors in our collection, most of them ended up not being challenging enough for both generators. Besides, despite the informal characterization reported in Appendix A, a more systematic characterization would be desirable for an extensive collection of errors.

APPENDIX A – Design error characterization

This appendix supplements Section 6.1 by presenting a more precise characterization of the design errors selected for our experimental evaluation.

Recall that our criteria for selecting design errors were the following: 1) errors should be chosen by analyzing their behavior from the perspective of a conventional generator; 2) all errors should be exposed by a conventional generator for at least one combination of parameters within the ranges defined for the experiment; 3) each error should exhibit a quite different behavior such that it could serve as an archetype for similar errors.

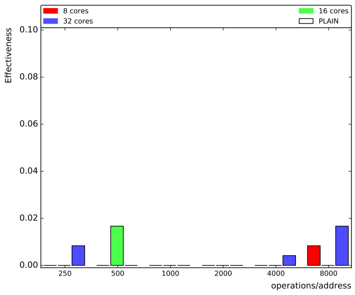
We have exploited two characteristics to assess the third criterion: how the probability of error detection behaves with the growing number of cores and how it behaves with the growing number of operations sharing the same location, which we call the *sharing level* of a test program.

Let us also remind that two operations *conflict* when they reference the same address and at least one of them is a store. Let us call the average number of conflicting operations in a test program its *conflicting level*. The choice of parameters for a generator determines the sharing level of the generated tests and imposes an upper bound on their conflicting level (since two loads never conflict).

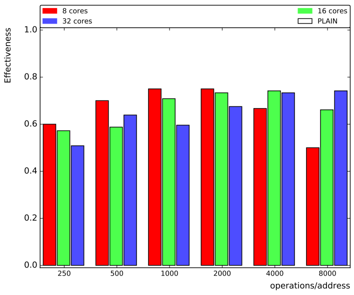
Figure 23 shows, for each error, a distribution for the probability of error exposure (measured with a conventional generator) as a function of generation parameters. Each distribution was built within the range of parameters adopted for the experiments. For each different core count ($\boldsymbol{p}$), it shows the probability of error detection for tests with same ratio between operation and location counts ($\boldsymbol{n/s}$). The sub-figures employ distinct scales to accommodate probabilities of quite different orders of magnitude, because errors F1, F4, and f29 are much more unlikely to be detected than the others.

Note that the behavior observed for the detection probability with respect to sharing level is completely different from one type of error to another. For instance, F2 can be easily detected in all verification scenarios, regardless of sharing level, i.e. its detection is largely independent of parameter setting. On the other hand, although error F3 can also be detected in all verification scenarios, its probability of detection can be improved (to a certain extent) by conveniently choosing the parameters so as to increase the sharing level. In contrast, note that errors F1, F4, and f29 cannot be detected in several verification scenarios. For errors F1 and f29, most scenarios leading to detection correspond to the highest sharing levels. However, for error

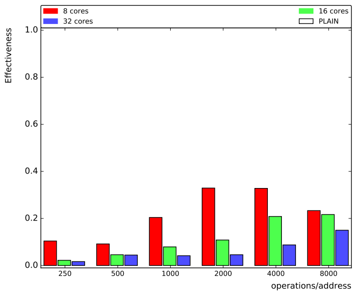
F4, whenever detection was observed, it was largely independent of sharing level.



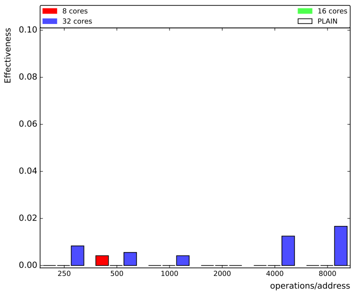
(a) F1



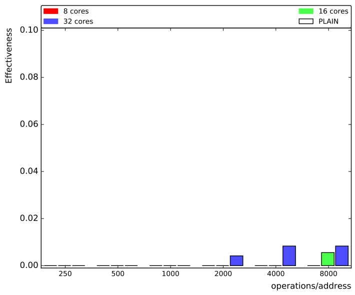
(b) F2



(c) F3



(d) F4



(e) f29

Figure 23: Sharing-level distributions for errors

To explain why some errors are sensitive to sharing level and others are not, we have to remind that some actions associated with the transitions of a cache controller's FSM are actual coherence actions and others not. For instance, errors affecting store serialization and single-writer multiple-read invariance (two main requirements for coherence) are exposed by inter-processor conflict. Besides, errors affecting the preservation of program order in relaxed consistency models are exposed by intra-processor conflict. Since the sharing level is an upper bound estimate for the conflicting level, an increase in sharing level is likely to expose errors affecting those requirements.

Notice that the behavior observed for the detection of probability with respect to core up-scaling is also completely different from one type of error to another. For instance, for errors F4 and f29, most scenarios leading to detection correspond to the highest core counts. This means that their detection tends to increase with up-scaling. For error F1, this effect is less dominant and, conservatively, we can assume that its detection is largely independent of core up-scaling. For error F2, there is no clear trend: its detection probability is largely independent of core up-scaling. Finally, error F3 becomes harder to expose with core up-scaling.

Since errors like F3 may become a challenge for pre-silicon verification under sheer core up-scaling, F3 deserves a detailed analysis to explain why its probability of detection decreases with increasing core counts. To do so, we first need to analyze how to expose F3. For this, we have to consider a few transitions of the protocol's FSM. The actions on transitions (E, I) and (M, I) are indistinguishable, although the actions on transitions (E, S) and (M, S) can be distinguished, since the latter induces a write-back action (but not the former). Since (as depicted in Table 1) F3 suppresses the transition (E, M), the transition (M, S) will never occur and the write-back action will be missing. Therefore, the missing action can only be detected if the cache line containing the block is *replaced* by a subsequent store that competes for that same line, so that a following load references the block just evicted and receives a stale copy from a lower hierarchical level. Now, let us analyze why replacements become less likely with core up-scaling. For a given test length, the higher the number of cores, the smaller the number of operations in each thread and, therefore, the smaller the number of locations *potentially* competing for the same (private) cache row. Unfortunately, the actual amount of competition of distinct locations for the the same cache row depends on cache configuration and effective address. Therefore, it cannot be controlled by the sequence generator. Since the test length for pre-silicon verification is limited to tens of thousands of operations and the number of cores keeps growing, this results in decreasing thread length and, therefore, decreasing potential for cache row competition. Thus, the remaining potential should be exploited as much as possible. That is why the adequate use of competition biasing constraints is crucial, making the address assigner (proposed in Chapter 4) a promising technique.

APPENDIX B – Comparison of the generators in supplementary scenarios

This appendix supplements Sections 6.3.1 and 6.3.2 with extra comparisons between the proposed and the conventional generator over the same verification scenarios, which corresponds to best exposure for only one of the them (the other is operating far from its best exposure).

Given the subset of verification scenarios restricted to 8 shared locations, Table 9 shows the percentage of them leading to error exposure. Note that, on average, CHAIN+ improved exposure by 9%. On the other hand, for a subset restricted to 32 shared locations, Table 10 shows that CHAIN+ improved exposure by 31%. Beside, CHAIN+ led to around 4 times more exposure in mutually-exclusive scenarios as compared to PLAIN+. In short, such results show that CHAIN+ led to superior error exposure, especially for large numbers of shared locations.

Table 9: Percentage of verification scenarios with potential for error exposure (s=8)

Exposed by	F1	F2	F3	F4	f29	avg
PLAIN+ . CHAIN+	7%	100%	87%	7%	7%	41%
PLAIN+ . not CHAIN+	7%	0%	0%	0%	7%	3%
CHAIN+ . not PLAIN+	20%	0%	13%	0%	0%	7%
PLAIN+	14%	100%	87%	7%	14%	44%
CHAIN+	27%	100%	100%	7%	7%	48%

Table 10: Percentage of verification scenarios with potential for error exposure (s=32)

Exposed by	F1	F2	F3	F4	f29	avg
PLAIN+ . CHAIN+	0%	100%	93%	0%	7%	40%
PLAIN+ . not CHAIN+	13%	0%	0%	13%	0%	5%
CHAIN+ . not PLAIN+	67%	0%	7%	20%	0%	19%
PLAIN+	13%	100%	93%	13%	7%	45%
CHAIN+	67%	100%	100%	20%	7%	59%

Tables 11 and 12 show the relative effectiveness when joint-exposure is restricted to verification scenarios with 8 or 32 shared locations. Note that CHAIN+ is more effective for two errors and less effective for a single error, even in the verification scenarios with 8 locations, where all errors are detected. In the scenarios with 32 locations, where only errors F2, F3, and f29 are jointly exposed, PLAIN+ is never superior to CHAIN+, except for F2 in the worst case.

Table 11: Improvement in effectiveness under joint exposure (s=8)

Errors	**Best**	**Worst**	**On average**	
F1	**1.00**	**1.00**	**1.00**	0%
F2	**1.19**	**0.91**	**1.00**	0%
F3	**3.83**	**0.50**	**2.03**	**+103%**
F4	**0.50**	**0.50**	**0.50**	-50%
f29	**2.00**	**2.00**	**2.00**	**+100%**

Table 12: Improvement in effectiveness under joint exposure (s=32)

Errors	**Best**	**Worst**	**On average**	
F1	——	——	——	——
F2	**1.21**	**0.95**	**1.04**	**+4%**
F3	**13.00**	**1.40**	**6.26**	**+526%**
F4	——	——	——	——
f29	**1.00**	**1.00**	**1.00**	0%

REFERENCES

ABTS, D.; SCOTT, S.; LILJA, D. J. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In: *IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*. [S.l.: s.n.], 2003. p. 10–pp. 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. et al. Verification of Transactional Memory in POWER8. In: *IEEE Design Automation Conference (DAC)*. [S.l.: s.n.], 2014. p. 58:1–58:6. ISSN 0738-100X.

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

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

AHARON, A. et al. Verification of the IBM RISC system/6000 by a dynamic biased pseudo-random test program generator. *IBM Systems Journal*, v. 30, n. 4, p. 527–538, Apr 1991. ISSN 0018-8670.

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.: s.n.], 2016. p. 552–559.

BIN, E. et al. Using a constraint satisfaction formulation and solution techniques for random test program generation. *IBM Systems Journal*, v. 41, n. 3, p. 386–402, Apr 2002. ISSN 0018-8670.

BINKERT, N. et al. The gem5 Simulator. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 39, n. 2, p. 1–7, Aug 2011. ISSN 0163-5964.

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 Int. Conf. Computer Aided Verification (CAV)*. [S.l.]: Springer Berlin Heidelberg, 2002, (Lecture Notes in Computer Science, v. 2404). p. 123–136. ISBN 978-3-540-45657-5.

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

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

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

FREITAS, L. S.; RAMBO, E. A.; SANTOS, L. C. V. dos. On-the-fly Verification of Memory Consistency with Concurrent Relaxed Scoreboards. In: *Design, Automation, and Test in Europe (DATE)*. [S.l.: s.n.], 2013. p. 631–636. ISBN 978-1-4503-2153-2.

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

HANGAL, S. et al. TSOtool: A program for verifying memory systems using the memory consistency model. *ACM SIGARCH Comp. Arch. News*, ACM, New York, NY, USA, v. 32, n. 2, p. 114–123, Mar 2004. ISSN 0163-5964.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. 5th. ed. [S.l.]: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X, 9780123838728.

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

LEWIN, D. et al. Constraint satisfaction for test program generation. In: *14th IEEE Int. Phoenix Conference on Computers and Communications*. [S.l.: s.n.], 1995. p. 45–48.

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

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

NAVEH, Y. et al. Constraint-based random stimuli generation for hardware verification. *AI magazine*, v. 28, n. 3, p. 13, 2007.

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

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

RAMBO, E.; HENSCHEL, O.; SANTOS, L. dos. On ESL verification of memory consistency for system-on-chip multiprocessing. In: *Design, Automation, and Test in Europe (DATE).* [S.l.: s.n.], 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 (CAV).* [S.l.]: Springer Berlin Heidelberg, 2006, (Lecture Notes in Computer Science, v. 4144). p. 503–516. ISBN 978-3-540-37411-4.

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

WAGNER, I.; BERTACCO, V. MCjammer: Adaptive Verification for Multi-core Designs. In: *Design, Automation, and Test in Europe (DATE).* [S.l.: s.n.], 2008. p. 670–675. ISSN 1530-1591.