Chaining and Biasing: Test Generation Techniques for Shared-Memory Verification

Gabriel A. G. Andrade, Marleson Graf, and Luiz C. V. dos Santos

***Abstract*—Since nondeterministic behavior is key to exposing shared-memory errors, nonsynchronized parallel programs are often used for verification and test of multicore chips. In the verification phase, however, the slow execution in a simulator requires nonconventional constraints for enabling error exposure with shorter programs. This paper proposes two novel techniques that build upon conventional random test generation for efficient shared-memory verification. The first technique exploits canonical dependence chains for constraining the random generation of instruction sequences so that the races induced at runtime are likely to raise the coverage of state transitions due to memory events conflicting at a same shared location. The second one exploits address space constraints for biasing random address assignment so that the competition of distinct shared locations for a same cache set can be controlled for raising the coverage of state transitions due to eviction events. We built generators relying on each of the proposed techniques, as well as on their combination, and we compared them to a conventional constrained random test generator for 8, 16, and 32-core architectures. Each of the four generators synthesized 1200 distinct test programs for verifying ten faulty designs derived from each of the three architectures (144 000 verification runs in total). For 32-core designs, the combination of the proposed techniques made at least 50% of the generation space capable of exposing errors, improved the median functional coverage by 44% and 83% at the two highest hierarchical levels, and reduced the average verification effort by one order of magnitude in many cases.**

***Index Terms*—Coherence, design aids, shared memory, single-chip multiprocessors, test generation, verification.**

I. 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 multiprocessors [1], [2]. Shared-memory verification is usually addressed at two levels of abstraction [4]: 1) the architectural verification of the cache coherence protocol and 2) the verification of its corresponding design representation. At the higher level, industry relies on some (incomplete) combination of simulation-based and formal approaches, because the complete formal verification of protocols is not practical, albeit recent approaches (e.g., [5]) devise ways to overcome the manual effort required by theorem proving and the scalability limitations of model checking. At the lower level, functional processor verification largely relies on simulation.

The literature reports three main simulation-based approaches.

1) *Functional processor verification frameworks* [6]–[8] provide unified mechanisms to synthesize tests that can check the design of an entire multicore chip, including the shared-memory subsystem.
2) *Protocol-based verification* [9], [10] relies on the state machine specifying the coherence protocol and tracks its states and transitions to guide test program generation.
3) *Memory-model verification* [11]–[13] exploits the memory consistency model (MCM) [3] for reducing the coupling between verification and implementation details.

Instead of directly solving a unified constraint satisfaction problem for the whole system [6]–[8], we reduced the scope of verification to the shared-memory subsystem, and we decomposed test generation into two main subproblems: 1) constrained random thread generation and 2) constrained random address assignment. The motivation for such decomposition lies in the fact that short random tests are unlikely to induce enough racy operations and sufficient eviction events for adequate coverage unless generation subtasks are properly constrained. The decomposition results from two properties of canonical dependence chains [14]: their ability to favor orderings leading to coherence events and their independence from the effective addresses assigned to the shared locations.

This paper proposes novel techniques that exploit nonconventional constraints on random test generation. The reduction in scope and the decomposition fostered the design of novel, specific algorithms to solve the subproblems instead of relying on generic solvers, and led to two main contributions.

1) A technique named *chaining* that exploits multiprocessor dependence chains for constraining thread generation.
2) A technique named *biasing* that exploits a partitioning of the shared locations for constraining their effective addresses.

This paper is organized as follows. Section II reviews related work. Section III shows the decomposition of the target problem. Section IV clarifies the key ideas. Sections V and VI

Manuscript received August 10, 2018; revised October 22, 2018 and December 10, 2018; accepted January 10, 2019. Date of publication January 21, 2019; date of current version February 24, 2020. This work was supported in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico and in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil (CAPES) - Finance Code 001. This paper was recommended by Associate Editor R. Drechsler. *(Corresponding author: Gabriel A. G. Andrade.)*

G. A. G. Andrade is with the Graduate Program in Automation and Systems Engineering, Federal University of Santa Catarina, Florianópolis 88040-900, Brazil (e-mail: gabriel.arthur@posgrad.ufsc.br).

M. Graf is with the Graduate Program in Computer Science, Federal University of Santa Catarina, Florianópolis 88040-900, Brazil.

L. C. V. dos Santos is with the Graduate Program in Automation and Systems Engineering, Federal University of Santa Catarina, Florianópolis 88040-900, Brazil, and also with the Graduate Program in Computer Science, Federal University of Santa Catarina, Florianópolis 88040-900, Brazil.

Digital Object Identifier 10.1109/TCAD.2019.2894376

provide examples and describe the algorithms solving the target subproblems. Section VII compares the proposed techniques with a conventional test generator. Section VIII draws the main conclusions and discusses future work.

II. Related Work

An MCM specifies rules defining the degree of program order relaxation and the extent of store atomicity [3]. There are two main testing approaches that rely on MCMs: 1) the combination of litmus test generation with checking of valid execution witnesses and 2) the coupling of random test generation and memory-model checking.

The first approach exploits the MCM for automated generation of litmus tests [17]–[19], i.e., short concurrent programs designed to stress certain MCM behaviors. The MCM declares which test outcomes are legal and which are not [19]. Each test is run thousands of times to provoke the behavior that the test characterizes [18]. Despite its success in finding subtle bugs when testing commercial architectures [17], this approach has limited coverage at design time [13].

In the second approach, memory-model checkers exploit the MCM for reducing the coupling between testing and implementation details. This paper by Hangal *et al.* [15] inspired many post-silicon checkers (e.g., [20]–[23]), which elaborated on their original idea. This allowed for more reusable checkers and extended post-silicon testing beyond race-free self-checking tests and toward more effective random tests with intensive data races. However, the claims that post-silicon checkers could be efficiently reused [15], [23] for presilicon verification only hold for their best effort versions, but not for their complete versions (offering verification guarantees) [20], [23], whose poor scalability with growing core counts severely limits their reuse at design time.

The literature reports two classes of presilicon *runtime* checkers based on memory models. A relaxed scoreboard [11] was proposed for keeping multiple valid events per entry. It employs an update rule that stores a new event after each write and removes events that become invalid after each read until an entry narrows down to a single memory event. Since it never reconsiders a previous decision, the technique admittedly may raise false negatives for a given test program. In contrast, another work [12] proposes the use of multiple verification engines (one per core) and a single global checker to build an axiom-based runtime checker with proven guarantees.

Industrial environments have been relying on random generators for processor verification since the mid-1980's. For instance, IBM's Genesys-Pro casts random test generation into a constraint satisfaction problem [7]. Albeit not originally intended to handle nondeterministic behavior, it was extended to allow the random generation of collision scenarios [6], because programs with intensive data races expose bugs faster [11], [15]. This observation has also fostered random test generation techniques specifically targeting the memory subsystem through memory-model checking, either for test [15], [20], [21], [23] or verification [11], [12].

As opposed to post-silicon testing, verification cannot afford long tests to achieve coverage goals. To reach similar goals with shorter tests, directed-test generation has been advocated [9], [10], [13]. In face of the growing number of cores, one of the keys to scalability is the decomposition of the state space. In MCjammer [9], 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 [10] 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. In McVerSi [13], Genetic Programming progressively improves the quality of the test suite. It relies on a crossover function that prioritizes memory operations contributing to nondeterministic behavior.

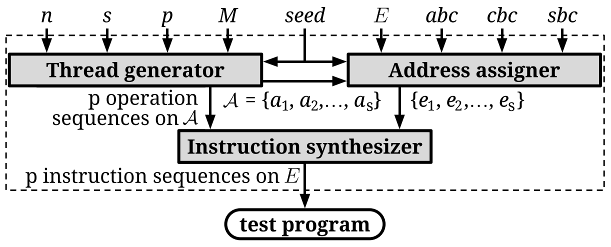

n
s
p
M
seed
E
abc
cbc
sbc
Thread generator
Address assigner
p operation sequences on $\mathcal{A}$
$\mathcal{A} = \{a_1, a_2, \ldots, a_s\}$
$\{e_1, e_2, \ldots, e_s\}$
Instruction synthesizer
p instruction sequences on E
test program

Fig. 1. How the generation flow can be decomposed into interacting engines.

Since directed-test generation often relies on some basic random generation engine, it can also benefit from improvements on constrained random test generation. As shown in an early work [16], there is unexploited room for improvement. This paper casts such early work into the proposed chaining technique, and it shows that, its combination with the novel biasing technique, leads to a generalized random test generation flow that is more effective than a conventional one.

III. Proposed Generation Flow

Fig. 1 shows the structure of the proposed generator.

Given the number of processors (p), the target numbers of memory operations (n) and shared locations (s) for the test program, a target mix of canonical dependence chains (M), and a random seed, the *thread generator* builds p sequences (threads) containing each n/p operations with references to locations in the set $\mathcal{A} = \{a_1, a_2, \ldots, a_s\}$. To build the sequences, it exploits canonical chains [14] to increase the probability of error exposure and to foster higher coverage.

Given the address space (E), the *address assigner* maps *locations* $a_1, a_2, \ldots, a_s$ to effective *addresses* $e_1, e_2, \ldots, e_s$. The mapping relies on three types of biasing constraints to enforce desirable properties: 1) the alignment biasing constraint (abc) specifies the intended address stride; 2) the sharing biasing constraint (sbc) specifies whether true sharing must be enforced or not; and 3) the competition biasing constraint (cbc) specifies how shared locations compete for cache sets.

The *instruction synthesizer* simply converts p sequences of memory *operations* referencing locations into p sequences of memory *instructions* referencing effective addresses. Besides, it also defines unique values to be written by store instructions, as required by most memory checkers [12], [15], [20], [23].

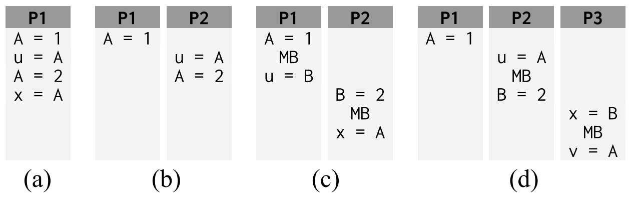


Fig. 2. Examples of chain Categories 0 (a), 1 (b), 2 (c), and 3 (d). Upper case letters denote variables in memory; lowercase letters, variables in registers.

IV. Key Ideas

This section shows how and why chaining and biasing are likely to increase coverage and error detection.

A. Exploitation of Chains for Thread Generation

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

Let us first informally introduce a few notions from [14], which are required for the examples (and will be formally defined in Section V-A). Two operations *conflict* if they collide at the same location and at least one of them is a store. Two operations from the same thread are in *significant program order* either if they conflict or if they are ordered by an MB. Two operations from distinct threads are in *significant conflict order* either if they are in conflict order or if they are colliding loads with an intervening store in conflict order with them. Different *categories* of canonical chains can be defined by such significant orderings, as illustrated in Fig. 2.

Fig. 2(a) shows a uniprocessor dependence chain with all operations in significant program order. Since nondeterministic behavior is key to the exposure of shared-memory errors [11], [13], [15], Fig. 2(b)–(d) illustrates multiprocessor dependence chains that form data races between threads, but where operations in a same thread must execute in significant program order. Races are formed when operations from distinct threads are in conflict order.

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

Our technique exploits canonical chains not for enforcing specific *consistency* rules, but for favoring proper *coherence* events instead. Fig. 3 shows the conceptual connection between a canonical chain and coherence events for different protocols. Note that, as the operations in the chain are executed, an intracore conflict leads to local requests that induce *distinct* transitions in the local FSM, while an intercore conflict leads to local and remote requests that also induce *distinct* transitions. Thus, the chain's *alternation* between intra and intercore conflicts tends to induce different transitions, which favors transition coverage. Since distinct protocols have similar responses for the same coherence transactions (except for a few transitions and write-back actions), this general property of a canonical chain makes the impact of our technique largely *independent* of the protocol implemented in a given design.

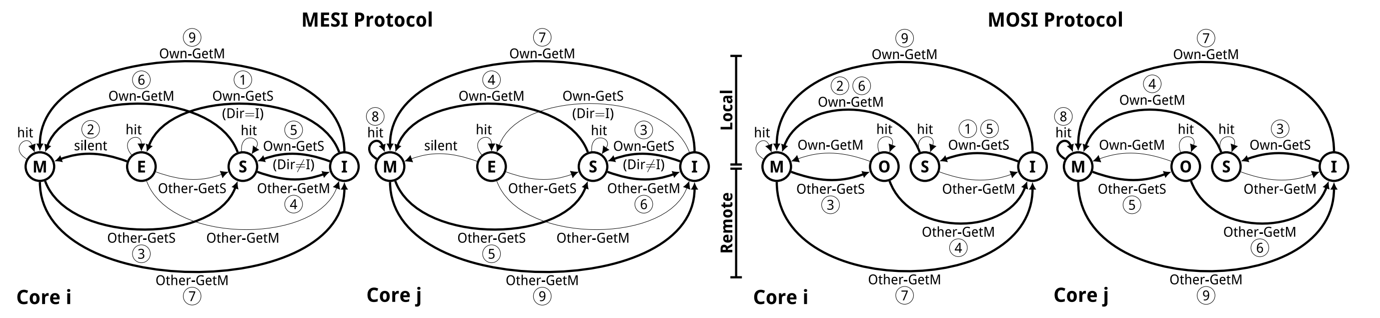
Chaining assumes the relaxation of program order for accesses to distinct locations only when it builds a *given* chain. Its focus on coherence does not restrain its applicability for four reasons.

1) The random selection of locations in distinct chains ensures that the stimulation of stronger orderings is not excluded from the generation space (albeit not favored in the scope of a given chain).
2) The checker is the guardian of memory model semantics (it may have to check orderings that are stronger than those relaxed for improving test quality).
3) Albeit the use of barriers does not improve coherence coverage, chaining preserves them for defining *general* significant orderings (instead of enforcing *stricter* conflict orderings, as in Fig. 3), because they enable the detection of consistency errors not tied to coherence mechanisms.
4) When complex fences (as in ARMv8 and Power8) replace simple barriers in canonical chains, our technique can handle sophisticated MCMs.

B. Exploitation of Constraints for Address Assignment

The main idea behind address assignment is the cbc. Let us illustrate that idea by means of an example. Fig. 4(a) shows a layout corresponding to a 32-bit address space, but where 26-bit block addresses are actually represented, because it assumes blocks with 64 bytes.

Suppose that the address range is limited to 4 MB, and it is partitioned into segments with 1 MB each, as depicted by the light gray boxes. Suppose, however, that the address space to be exploited by the generator is constrained to only 2 KB in total, and it is uniformly distributed over the partitions in *useful* subranges with 512 B each, as depicted by the dark gray boxes. Note that, due to the 1 MB partitioning, the block addresses can accommodate indices for identifying up to 2^{14} distinct cache sets. In spite of that, the constraint imposed on useful subranges leads to a bound of 8 for the number of different index identifiers. Besides, the constraint imposed on the amount of partitions of the full address range leads to a bound of four for the number of distinct tags identifiers. This way of restricting the address space into a few useful chunks



$$\underset{①}{\mathbf{L_a^i}} \prec_{po} \underset{②}{\mathbf{S_a^i}} \le \underset{③}{\mathbf{L_a^j}} \prec_{po} \underset{④}{\mathbf{S_a^j}} \le \underset{⑤}{\mathbf{L_a^i}} \prec_{po} \underset{⑥}{\mathbf{S_a^i}} \le \underset{⑦}{\mathbf{S_a^j}} \prec_{po} \underset{⑧}{\mathbf{L_a^j}} \le \underset{⑨}{\mathbf{S_a^i}}$$

Fig. 3. How a canonical chain improves the coverage of coherence events independently of the protocol implemented in a given design. (L_a^i and S_a^j stand for load and store operations issued by processors i and j for some location a. $\prec_{\text{po}}$ and $\le$ denote, respectively, program and execution orders).

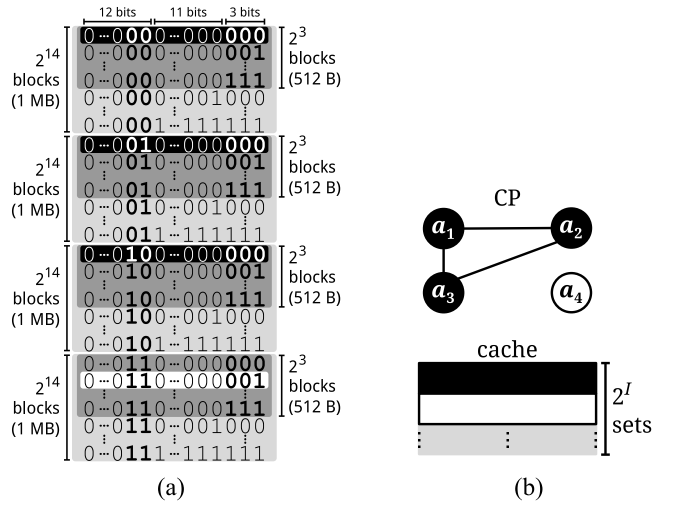


Fig. 4. Address assignment for four shared locations under address space constraints (assuming blocks with 64 bytes). A block address consists of a 12-bit tag and a 14-bit index where 11 bits are fixed but 3 may vary. (a) Address space constraints. (b) CP induced by cbc = (2, 3).

(as depicted by the dark gray boxes inside the light gray ones) is sometimes exploited (e.g., [13]) as a static address space constraint for fostering replacement events.

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

For a given number of locations, a competition biasing constraint cbc $= (\kappa, \chi)$ specifies patterns where all locations map to one out of κ cache sets, and at most χ locations map to the same cache set. For instance, the address assignment illustrated in Fig. 4(a) was induced by cbc $= (2, 3)$.

For a given cbc, there are different ways in which locations may compete for cache blocks. They can be seen as patterns, each representing distinct ways of partitioning the set of locations, as follows. Let set(a_i, a_j) denote that a_i and a_j compete for the same cache set and let C be the collection of such locations, i.e., $C = \{(a_i, a_j) : a_i, a_j \in \mathcal{A} \wedge \text{set}(a_i, a_j)\}$. Since C is an equivalence relation, a CP is a collection CP $= \{A_x\}$ forming a partition of the set $\mathcal{A}$ where each set A_x is an equivalent class (EC) induced by C.

Fig. 4(b) illustrates the notion of CP for a scenario with four locations and a cache with 2^I sets. For convenience of illustration, each EC is represented as a clique of an undirected graph. Therefore, given a cbc $= (\kappa, \chi)$ inducing a CP, κ and χ could be seen as clique cover and chromatic numbers, respectively.

For a given number s of locations, a same cbc may induce a collection of CPs. Thus, to avoid limiting the verification space, the address assigner must be able to randomly select one among them. Notice, however, that not all pairs (κ, χ) represent feasible constraints. Fortunately, the collection of all feasible pairs for a given s can be easily precomputed.

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

Another desirable property of an address assignment is specified by the sbc. The address assignment in Fig. 4(a) can be used to illustrate this notion. Note that the locations competing for the same cache set have distinct block addresses: despite the same index, their tags are all different. Since unrelated shared variables are not stored in the same memory block, such assignment precludes false sharing. Actually, the sbc is a Boolean value specifying if true sharing must be enforced or not.

Yet another desirable property is specified by the abc. The abc is a natural number specifying that all effective addresses must be aligned to 2^{abc}-byte boundaries. For instance, if we enforce the alignment to 2^6 bytes, the six offset bits implicit in Fig. 4(a) must be zero for all effective addresses to be exploited by the generator.

The motivation for mapping locations to effective addresses lies in the control of replacement events. For instance, the alternation between cbcs enabling and disabling block replacement tends to avoid revisiting the same state transition, which favors coverage and the probability of exposing design errors.

When multiple data races for distinct locations do not interfere with each other, the state transitions induced by each of them are quite similar. However, when they are coupled due to address space constraints (e.g., if two distinct locations are allocated in the same memory block or if they map to the same cache set), the induced state transitions tend to be rather different. That is, why the combination of chaining and biasing is likely to lead to higher coverage than the application of each technique alone, as reported in Section VII.

The next sections formalize the notions above and propose algorithms to solve the two main subproblems.

V. Thread Generation

This section describes how dependence chains are exploited when generating sequences of operations for distinct threads.

A. Formalization of the Main Notions

Let $\mathcal{A}$ be the set of all locations referenced by all operations in a test program. To specify that an operation O_j is issued by a processor i and makes a reference to some location $a \in \mathcal{A}$, we write $(O_j)^i_a$. We replace O by L or S to specify that the operation is either a load or a store. As shorthand notation, we sometimes drop either a superscript or a subscript. Let $O_j \prec_{\text{po}} O_m$ denote that two operations are in program order.

A test program may induce many executions with distinct outcomes, each inducing a memory behavior. Every valid behavior must satisfy a partial order $\leq$ on the set of memory operations. From the *program order* $\prec_{\text{po}}$, an MCM specifies the allowed *execution order* $\leq$. The literature reports axioms formally defining $\leq$ for distinct MCMs [12], [20], [21].

To formally define chain categories, we adopted a description idiom where simple barriers are used for enforcing program order between operations to distinct locations. Despite the idiom's simplicity, chains can be built with more complex fences without loss of generality, as explained in Section IV-A.

Definition 1: 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. We say that two operations are in *significant program order*, written $O_j \prec_{\text{spo}} O_m$, iff one of the following holds: $(O_j)_a \prec_{\text{po}} (O_m)_{b=a}$ or $(O_j)_a \prec_{\text{po}} MB \prec_{\text{po}} (O_m)_{b\neq a}$.

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

Definition 3: We say that two operations are in *significant conflict order*, written $O_j \leq_{\text{sco}} O_m$, iff $(L_j)^i_a \leq (S_m)^k_a \vee (S_j)^i_a \leq (L_m)^k_a \vee (S_j)^i_a \leq (S_m)^k_a \vee (L_j)^i_a \leq S^x_a \leq (L_m)^k_a$.

Definition 4: A chain is a sequence $X \prec U \prec \cdots \prec V \prec Y$ whose endpoints X and Y are memory operations, but $U, \ldots, V$ may represent either memory operations or MBs. The relation $\prec$ between two successive elements denotes one of the relations $\prec_{\text{po}}, \prec_{\text{spo}}, \leq, \leq_{\text{co}}$, or $\leq_{\text{sco}}$. Let $\{A \prec B \prec\}_*$ denote zero or more pattern occurrences in the chain, and let $\{A \prec B \prec\}_+$ denote one or more.

To formalize the canonical chains specified by Gharachorloo [14], we use L, S, and O to denote *types* of memory operations (respectively, load, store, any). We assume that distinct operation instances of each type will be used for building the chain.

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

Category 1: $S^i_a \leq L^j_a \prec_{\text{po}} L^j_a$ or $S^i_a \leq L^j_a \prec_{\text{po}} S^j_a$, where $i, j \in \{1, \ldots, p\}$ and $i \neq j$.

Category 2: $O^i_a \prec_{\text{spo}} \{O^i_b \leq_{\text{sco}} O^j_b \prec_{\text{spo}}\}_+ O^j_a$, where $i, j \in \{1, \ldots, p\}$, $i \neq j$ and b is arbitrary.

Category 3: $S^i_a \leq_{\text{sco}} L^j_a \prec_{\text{spo}} \{O^j_b \leq_{\text{sco}} O^k_b \prec_{\text{spo}}\}_+ L^k_a$, where $i, j, k \in \{1, \ldots, p\}$, $i \neq j$, $j \neq k$, and b is arbitrary.

B. Problem Formulation

Let p, n, and s be, respectively, the number of processors, the number of memory operations, and the number of distinct shared locations. Let M be a mix of patterns, which specifies the target fraction of chains from each category. Let $\mathcal{C}_\gamma$ denote the set of chains from category γ in a given test program and let $\mathcal{C} = \cup^3_{\gamma=0} \mathcal{C}_\gamma$ 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.

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

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

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; and the fourth, that the obtained proportions should reach the target mix.

C. Example of How to Build Solution

The proposed thread 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.

Fig. 5 shows how the proposed generator inserts a chain from Category 2 in a test program under construction. Suppose that two chains from Category 0 were already created [Fig. 5(a)]. Assume that Category 2 was randomly selected as target. The generator first tries to build a minimal chain from that category, but might randomly decide to extend it as far as it does not violate any constraints. To build it, the generator randomly selects a processor (say P2) and adds to the first free slots of the respective thread two memory operations with an intervening barrier [Fig. 5(b)]. The memory operation types are randomly selected [in Fig. 5(b), 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 Fig. 5(c), 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,

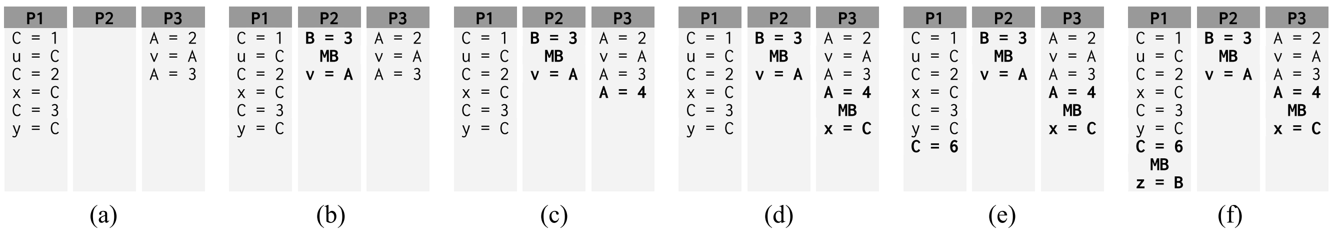

P1 P2 P3
(a) P1: C = 1, u = C, C = 2, x = C, C = 3, y = C; P2: ; P3: A = 2, v = A, A = 3
(b) P1: C = 1, u = C, C = 2, x = C, C = 3, y = C; P2: B = 3, MB, v = A; P3: A = 2, v = A, A = 3
(c) P1: C = 1, u = C, C = 2, x = C, C = 3, y = C; P2: B = 3, MB, v = A; P3: A = 2, v = A, A = 3, A = 4
(d) P1: C = 1, u = C, C = 2, x = C, C = 3, y = C; P2: B = 3, MB, v = A; P3: A = 2, v = A, A = 3, A = 4, MB, x = C
(e) P1: C = 1, u = C, C = 2, x = C, C = 3, y = C, C = 6; P2: B = 3, MB, v = A; P3: A = 2, v = A, A = 3, A = 4, MB, x = C
(f) P1: C = 1, u = C, C = 2, x = C, C = 3, y = C, C = 6, MB, z = B; P2: B = 3, MB, v = A; P3: A = 2, v = A, A = 3, A = 4, MB, x = C

Fig. 5. How the generator inserts a chain from Category 2 in a program, following steps (a) to (f). Uppercase letters denote variables in memory; lowercase letters, variables in registers.

the next memory operation becomes an endpoint and must conflict with the chain's starting point). Suppose 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 [Fig. 5(d)]. To accomplish the extension, a processor is randomly selected (say P1) and an operation conflicting with the previous one in the chain is inserted [Fig. 5(e)]. Again, the generator must decide whether the chain should be 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 Fig. 5(f), that endpoint *must* be a load from location B]. The resulting chain is represented in bold.

D. Algorithm

Let τ^i denote the thread assigned to processor *i*. Each thread consists of n/p slots. We write $\tau^i[x]$ 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 τ^i, which is denoted as av^i.

Operations can be selected from three different types: 1) load (*L*); 2) store (*S*); or 3) MB. To cope with the target proportion of chains specified for some category γ, i.e., $M[\gamma]$, our algorithm tracks the number of all slots available for that category, which is denoted as $C[\gamma]$.

Definition 5: Let $i, j, k \in \{1, 2, \ldots, p\}$ with $i \neq j, j \neq k, i \neq k$. We say that a minimal chain from category γ is feasible, written $\phi_{\min}(\gamma)$, iff one of the following holds for some i, j, k:

1) $\gamma = 0 \wedge C[\gamma] \geq 1 \wedge av^i \geq 1$;
2) $\gamma = 1 \wedge C[\gamma] \geq 3 \wedge av^i \geq 1 \wedge av^j \geq 2$;
3) $\gamma = 2 \wedge C[\gamma] \geq 4 \wedge av^i \geq 3 \wedge av^j \geq 3$;
4) $\gamma = 3 \wedge C[\gamma] \geq 5 \wedge av^i \geq 1 \wedge av^j \geq 3 \wedge av^k \geq 3$.

Definition 6: Let $X = \{x \in \mathcal{Z}_+ | x \leq n/p - av^i\}$ be the set of slots from thread τ^i which are filled with operations and let $X_{x>m} = \{x \in X | x > m\}$. The last slot of a thread τ^i containing a reference to location *a* is

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

Algorithm 1 shows the technique's top-level routine.[1] It reserves n/p slots for the operations in each thread (line 4) and evaluates the number of slots available for operations belonging to chains of each category (line 6). Then it performs chain generation (lines 7–14), i.e., it builds as many chains as feasible by randomly selecting a category and a location for the conflicting endpoints of each chain. The building of a chain starts if a minimal chain from the selected category is feasible (line 9), otherwise that category is made unavailable (line 13).

Algorithm 1 Thread-Generator (p, n, s, M)

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

Algorithm 2 describes the routine that builds a chain from category γ whose endpoints conflict at location *a*. We assume that $\mathcal{A}$, *C*, τ^i, and av^i all have global scope. 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, it 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). The precondition $\phi_{\min}(\gamma)$ ensures that the sets in lines 2, 12, 13, 19, 20, and 36–38 are all nonempty. 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 implementing the fourth clause of Definition 3). Their pseudocode is omitted for simplicity, but can be found in [16].

It can be shown that the overall worst-case running time complexity of the proposed thread generation algorithm is $O(s+p) + O(p\ n)$, i.e., $O(p\ n+s)$. In contrast, the worst-case complexity of a conventional generator is $O(p+n)$.

VI. Address Assignment

This section describes how biasing constraints are exploited when converting locations into effective addresses.

[1]Our implementation finds an approximate solution to Problem 1. Essentially, it relaxes Constraint 4 when needed to ensure Constraints 1–3.

Algorithm 2 Routine Chain (γ, a)

```
1: if $\gamma = 0$ then
2:     $i \leftarrow$ random $\{x \in \{1, 2, \ldots, p\} \mid av^x \geq 1\}$
3:     $\lambda \leftarrow$ random $\{1, 2, \ldots, av^i\}$
4:     while $\lambda \geq 1 \wedge av \geq 1 \wedge C[\gamma] \geq 1$ do
5:         if $max_a^i \neq 0$ and $\tau^i[max_a^i]$ is $L$ then
6:             $op \leftarrow S$
7:         else
8:             $op \leftarrow$ random $\{L, S\}$
9:         insert$(\gamma, op_a^i)$
10:        $\lambda \leftarrow \lambda - 1$
11: if $\gamma = 1$ then
12:     $j \leftarrow$ random $\{x \in \{1, 2, \ldots, p\} \mid av^x \geq 2\}$
13:     $i \leftarrow$ random $\{x \in \{1, 2, \ldots, p\} \mid av^x \geq 1 \wedge x \neq j\}$
14:     insert$(\gamma, S_a^i)$
15:     insert$(\gamma, L_a^j)$
16:     $op \leftarrow$ random $\{L, S\}$
17:     insert$(\gamma, op_a^j)$
18: if $\gamma = 2$ then
19:     $i \leftarrow$ random $\{x \in \{1, 2, \ldots, p\} \mid av^x \geq 3\}$
20:     $j \leftarrow$ random $\{x \in \{1, 2, \ldots, p\} \mid av^x \geq 3 \wedge x \neq i\}$
21:     $op \leftarrow$ random $\{L, S\}$
22:     insert$(\gamma, op_a^i)$
23:     insert$(\gamma, MB^i)$
24:     $b \leftarrow$ random $\mathcal{A} - \{a\}$
25:     $op \leftarrow$ random $\{L, S\}$
26:     insert$(\gamma, op_b^i)$
27:     $op \leftarrow$ sco $(\gamma, op, b, i)$
28:     insert$(\gamma, op_b^j)$
29:     $\lambda \leftarrow$ random $\{0, 3, 6, \ldots, C[\gamma]\}$
30:     if $\lambda \neq 0$ then
31:         $j \leftarrow$ extend$(\gamma, a, j, \lambda)$
32:     insert$(\gamma, MB^j)$
33:     $op \leftarrow$ random $\{L, S\}$
34:     insert$(\gamma, op_a^j)$
35: if $\gamma = 3$ then
36:     $j \leftarrow$ random $\{x \in \{1, 2, \ldots, p\} \mid av^x \geq 3\}$
37:     $i \leftarrow$ random $\{x \in \{1, 2, \ldots, p\} \mid av^x \geq 3 \wedge x \neq j\}$
38:     $h \leftarrow$ random $\{x \in \{1, 2, \ldots, p\} \mid av^x \geq 1 \wedge x \neq j \wedge x \neq i\}$
39:     insert$(\gamma, S_a^h)$
40:     insert$(\gamma, L_a^i)$
41:     insert$(\gamma, MB^i)$
42:     $b \leftarrow$ random $\mathcal{A} - \{a\}$
43:     $op \leftarrow$ random $\{L, S\}$
44:     insert$(\gamma, op_b^i)$
45:     $op \leftarrow$ sco $(\gamma, op, b, i)$
46:     insert$(\gamma, op_b^j)$
47:     $\lambda \leftarrow$ random $\{0, 3, 6, \ldots, C[\gamma]\}$
48:     if $\lambda \neq 0$ then
49:         $j \leftarrow$ extend$(\gamma, a, j, \lambda)$
50:     insert$(\gamma, MB^j)$
51:     insert$(\gamma, L_a^j)$
```

A. Problem Formulation

Assume that an N-bit effective address consists of three fields: 1) a block *offset* field with O bits (i.e., 2^O is the number of bytes in a cache block); 2) an index field with I bits (i.e., the cache has 2^I sets); and 3) a *tag* field with $T = N - O - I$ bits.

Let $\mathcal{CP}_{\kappa,\chi,s}$ be the collection of CPs induced by a given cbc $= (\kappa, \chi)$ and a given number of locations s. Let *random* $\mathcal{CP}_{\kappa,\chi,s}$ denote the random selection of a pattern from that collection. 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 bits. 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.

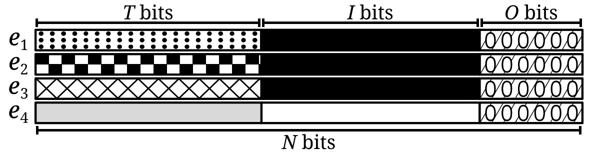


Fig. 6. How effective addresses are constrained by abc = 6 (block alignment), sbc = true (true sharing), and cbc = (2, 3).

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

1) $\{A_x\} =$ random $\mathcal{CP}_{\kappa,\chi,s}$;
2) $\forall a_i \in \mathcal{A}\colon\ \alpha(a_i).\mathcal{T} = \text{random}[0, 2^T - 1]$;
3) $\forall a_i \in \mathcal{A}\colon\ \alpha(a_i).\mathcal{I} = \text{random}[0, 2^I - 1]$;
4) $\forall a_i \in \mathcal{A}\colon\ \alpha(a_i).\mathcal{O} = (\text{random}[0, 2^{O-\text{abc}} - 1]) \times 2^{\text{abc}}$;
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) $a_i, a_j \in A_x \wedge \text{sbc} \Rightarrow \alpha(a_i).\mathcal{T} \neq \alpha(a_j).\mathcal{T}$.

Note that Conditions 1–4 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.

B. Example of How to Build Solution

Let us assume the address space constraints in Fig. 4(a) and the CP in Fig. 4(b). Given abc = 6 (block alignment), sbc = true (true sharing), and a CP randomly selected from those induced by cbc = (2, 3), Fig. 6 shows how the effective addresses are enforced to comply with such biasing constraints when assigning binary patterns to distinct address fields.

Remember that an EC from a CP corresponds to locations mapping to the same cache set. To enforce the CP, the same binary pattern must be assigned to the index field for elements from the same EC (in black) and a distinct binary pattern must be assigned to that field for elements from a different EC (in white). To enforce true sharing, distinct binary patterns must be assigned to the tag field for elements from the same EC, because this ensures that they will not lie in the same block (since their memory block addresses are different). Finally, to enforce block alignment, the six least significant bits from the offset field are set to zero (the field in gray remains unconstrained).

C. Algorithm

Our algorithm relies on a routine that iteratively enumerates all CPs induced by a given cbc, randomly selects one of them, and returns it. Enumeration is viable because a few tens of locations are sufficient for the short tests required for efficient verification (see Section VII-A).

Let us now explain how the intended mapping is built from the selected CP. 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 an EC 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 an EC, location a is mapped to address e. To enforce injection, $\mathcal{E}$ must be reduced to $\mathcal{E} \backslash \{e\}$ for the next iteration, since this precludes the reuse of e for future mappings. Assume that, while iterating over the ECs of a CP, the index i is assigned to some EC. 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 ECs (Problem 2 and Condition 5), $\mathcal{E}$ must be reduced to $\mathcal{E}\backslash\mathcal{E}_i$ for the next iteration, because this precludes the use of any address with the same index for future ECs. Finally, assume that, while iterating over the locations of an EC, 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 an EC. Let $\mathcal{E}_{i,t}$ be 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 EC (Problem 2 and 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 from the same EC, but not for locations from other ECs. Since $e \in \mathcal{E}_{i,t}$, the reduction ends up by also enforcing injection in this case.

To ensure that no location is left unmapped (while enforcing injection iteratively), the available address space must contain at least as many addresses with same index *and* distinct tags as the number of locations in A_x when true sharing is enforced. Otherwise, only the first condition is required, as follows.

Definition 7: Let $\mathcal{E}_i^\tau \subset \mathcal{E}_i$ be the subspace of the available addresses induced by the same index i and different tags. Given an EC A_x, an sbc, and the available address space $\mathcal{E}$, the set of candidate indices is

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

Algorithm 3 relies on the notions formalized above to describe the top-level routine of the proposed address assigner. Line 1 randomly selects a CP. 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 EC of the selected CP. Line 4 randomly selects a candidate index for the current EC. 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 ECs. Lines 6–14 iterate over each location a of the current EC. Line 7 randomly selects an effective address e with the index selected for the current EC. 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 EC (under true sharing). Both lines 11 and 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.

It can be shown that the worst-case complexity of the algorithm becomes $O(s \times |\mathcal{CP}_{\kappa,\chi,s}| \times \log|\mathcal{CP}_{\kappa,\chi,s}| + s \times |E|)$.[2]

VII. Experimental Evaluation

This section compares generators built with the proposed techniques to a conventional random test generator.

[2]Indeed, the last term of the complexity can be reduced from $O(s \times |E|)$ to $O(s^2)$ when the external loop (line 3) visits the ECs of a given CP in order of nonincreasing cardinalities.

Algorithm 3 Address-Assigner (abc, cbc, sbc, s, E)

```
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 $\alpha$
```

A. Experimental Set Up

Designs were derived for 8-, 16-, and 32-core architectures.[3] We relied on the pseudocode described in [12] to implement the checker and on the gem5 infrastructure [24] for simulation (*O*3, *Ruby*, and *simple* as CPU, memory, and interconnect models). We selected a 3-level MESI directory protocol that defines requests for read-only (GETS) and read-write (GETX and UPG) permissions, as well as eviction notifications for dirty blocks (PUTX). The hierarchy consisted of private L0 (split) caches, private L1 (unified) caches, and shared L2 cache, with 4 KB (directed-mapped), 64 KB (2-way), and 2 MB (8-way), respectively, all operating with the same block size (64 bytes) and the same replacement policy (LRU).

We compared four generators.

1) PLAIN− is a conventional generator, which is similar to the ones used for memory-model checking (e.g., [11] and [15]). The sequence of operations forming a thread is obtained by randomly selecting the location and the type of each operation independently of the choice made for other operations belonging to the same or some other thread. Address assignment is unconstrained (except for the obvious injection requirement), i.e., binary patterns are randomly selected from a specified address subspace. Since we could not find an implementation in the public domain, we relied on the pseudocode reported in [25] to implement our own prototype.
2) PLAIN+ is similar to PLAIN−, but the constrained address assignment technique (proposed in Section VI) replaces the conventional unconstrained assigner.
3) CHAIN− exploits the chain-based sequence generator from Section V, but relies on a conventional unconstrained address assigner.
4) CHAIN+ combines both mechanisms proposed in this paper.

All generators have common program parameters (p, n, s) and a common parameter for random generation (seed). However, each generator has specific parameters tied to its inner mechanism. PLAIN− and PLAIN+ rely on *instruction* mixes specifying the target proportions of load, store, and members (whose values were inspired by a related work [26]).

[3]In all designs, the adopted architecture was SPARC.

TABLE I
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	0.4	0.6	0	0
0.48	0.48	0.04	0	1	0	0
0.66	0.30	0.04	0	0.8	0.2	0
0.80	0.16	0.04	0	0.8	0	0.2

CHAIN− and CHAIN+ rely on *category* mixes specifying target proportions of chain categories (whose values were obtained empirically). When randomly selecting an operation for a chain, the generator employed the following probabilities: 0.75 for loads and 0.25 for stores. Table I shows the target mixes. Finally, PLAIN+ and CHAIN+ have common parameters for specifying biasing constraints.

To verify each design, a distinct test suite was synthesized with each generator. We compared the generators for a same setting of the common program parameters by letting the others vary within predefined ranges. 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 parameters, we relied on values reported from industrial environments [7], [20], [23]. Tests for post-silicon usage have hundreds of thousands of operations [20], [23] and a few hundreds of shared locations [20]. Tests for presilicon usage typically have tens of thousands of operations [7]. Thus, 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 operation conflict required by the best post-silicon practices. Each generator synthesized a distinct suite with 1200 tests per architecture by exploring 15 random seeds $(1, 2, 3, \ldots, 15)$, 4 target mixes, 4 amounts of shared locations $(s = 4, 8, 16, 32)$, and 5 program sizes $(n = 1k, 2k, 4k, 8k, 16k)$, with k standing for 2^{10} operations. We constrained the shared locations within a 2^{25} address subspace.

For the generators PLAIN+ and CHAIN+, each verification scenario was constrained by a single cbc, exactly the same for both. For the experiments reported in this paper, we selected cbc $= (1, s)$, because it fosters eviction as much as possible for the adopted range of s, allowing an evaluation of the proposed techniques that is largely independent of the associativity adopted at each hierarchical level. Besides, all addresses were aligned to the block (abc $= 2^6$) and true sharing was always enforced (sbc = true).

Since design errors are accidental, there is no standard collection of typical bugs that could be used as benchmark. Unfortunately, the errors reported in the literature are often tied to specific consistency models or are not described in sufficient detail to be properly reproduced. Therefore, as in many related works, we also had to rely on artificially injected errors to challenge the generators. We deliberately adopted coherence errors that are largely independent of consistency model and are easily reproducible. To build faulty designs, we used ten types of errors affecting cache controllers at levels L0, L1, and L2, as described in Table II, which indicates modified (white) and new (gray) transitions. Errors were classified according to the violation of either data value (DV) or single-writer-multiple-reader (SWMR) invariants. We assumed that an error is systemic, i.e., it results in replicas in all controllers at the same level. From a correct design, we derived ten *distinct* designs, each containing a *single* type of error.

For every architecture and every generator, we applied each test to ten designs, each containing an error from a different type (144 000 runs in total). Then we determined whether or not an error was exposed for each of them. We say that a test program *exposes* a design error if it leads the checker to detect a *violation* of the memory model. To avoid that false negatives (positives) could underestimate (overestimate) error exposure, we used a presilicon runtime checker with verification guarantees [12].

Run times were measured on an HP xw8600 Workstation (Intel Xeon E5430, 2.66 GHz) with 8 GB of main memory.

B. Metrics

To evaluate the *potential for error exposure* as measured by our experimental results, we determined whether or not a generator is likely to synthesize a test exposing the error under each (p, n, s) setting, and we obtained the fraction of all settings with potential for exposure (the higher the fraction, the smaller the sensitivity to parameter choice). Given two generators, to correlate their potentials for error exposure, we determined whether both, one, or none is likely to expose an error under the *same* (p, n, s). We adopted the following procedure for each type of error and every scenario. For a given (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 that scenario as "exposing." To correlate the potentials of a pair of generators, say G and g, we adopted the following procedure for each type of error. For each scenario (p, n, s), we checked whether or not both generators have marked it as exposing for a given error. If so, we labeled (p, n, s) as a scenario of *joint exposure* (written $G.g$). If not, we labeled it as a scenario of *mutually exclusive exposure*, depending on whether it was an exposing scenario either for G (written $G.\overline{g}$) or for g (written $g.\overline{G}$). Otherwise, it was labeled as a *joint nonexposure* scenario (written $\overline{G}.\overline{g}$). Finally, for a given type of error, we computed the fraction of all scenarios corresponding to each label.

To evaluate the *effectiveness* of a generator in exposing a given type of error under a scenario $v = (p, n, s)$, we measured the fraction $\varepsilon(v)$ of all tests induced by v for which violations were detected. Assuming sufficient sampling, this fraction can be interpreted as the probability of a generator to expose that type of error when the generation parameters are set to $v = (p, n, s)$. To calculate the average effectiveness for designs containing the same type of error, we took the arithmetic mean over the collection of all scenarios.

To estimate the *verification effort* of running the random tests induced by a scenario $v = (p, n, s)$ in an attempt to expose an error, we combined the effectiveness and the average test run times measured for a given design in scenario v. Let $T = \{T_i\}$ be the collection of tests induced by v and let t_i be the measured runtime for test T_i. Let $T^0 \cup T^1$ be a

TABLE II
TYPES OF ARTIFICIAL DESIGN ERRORS (FOR REPRODUCIBILITY, THE NAMES OF STATES, EVENTS, AND ACTIONS MATCH THOSE IN GEM5'S INFRASTRUCTURE)

	ID	Level	Current state	Input event	Next state	Output action
SWMR	D0	L1	IS	Data_Exclusive	EE instead of E	preserved
			I	WriteBack	I	popL0RequestQueue
			SS	WriteBack	SS	popL0RequestQueue
			M_I	WriteBack	M_I	popL0RequestQueue
			SINK_WB_ACK	WriteBack	SINK_WB_ACK	popL0RequestQueue
			EE	WriteBack	MM	same as in (E, MM)
	D1	L1	E	L0_Invalidate_when_INV_UPG_GETS instead of L0_Invalidate_Else	E_ILO	preserved
			E	L0_Invalidate_when_GETX	EE	none
			I	WriteBack	MM	same as in (E, MM) plus allocateCacheBlock
			S	WriteBack	S	popL0RequestQueue
	D2	L1	E	L0_Invalidate_when_INV_UPG_GETS instead of L0_Invalidate_Else	E_ILO	preserved
			E	L0_Invalidate_when_GETX	EE	none
			I	WriteBack	I	popL0RequestQueue
			S	WriteBack	S	popL0RequestQueue
	D3	L1	E	L0_Invalidate_when_INV_UPG_GETX instead of L0_Invalidate_Else	E_ILO	preserved
			E	L0_Invalidate_when_GETS	EE	none
			SS	WriteBack	SS	popL0RequestQueue
			S	WriteBack	S	popL0RequestQueue
DV	D4	L0	E	Store	M	dirty=1 precluded in store_hit
	D5	L1	M_ILO	L0_DataAck	EE instead of MM	preserved
	D6	L1	MM	Load	E instead of M	preserved
	D7	L2	MT	L2_Replacement	MCT_I instead of MT_I	preserved
	D8	L1	E_ILO	L0_DataAck	MM	writeDataFromL0Response precluded
	D9	L1	IS_I	DataS_fromL1	I	writeDataFromL2Response precluded

TABLE III
MEDIAN IMPROVEMENT IN COVERAGE OVER THE ENTIRE GENERATION SPACE

p	Chaining CHAIN- w.r.t PLAIN-			Biasing PLAIN+ w.r.t PLAIN-			Chaining and Biasing CHAIN+ w.r.t PLAIN-		
	L0	L1	L2	L0	L1	L2	L0	L1	L2
8	1.05	0.96	1.02	1.11	1.68	3.39	1.21	1.86	3.46
16	1.00	0.96	1.00	1.12	1.72	3.29	1.35	1.92	3.33
32	1.00	0.96	1.00	1.19	1.67	3.67	1.44	1.83	3.67

partition of T, where $T^0 = \{T_j{}^0\}$ and $T^1 = \{T_i{}^1\}$ are, respectively, the tests that do not expose an error and those that do. Let ε be the effectiveness measured in that scenario. Let us assume that all tests that do not expose an error (under a same scenario) take essentially the same time $\widehat{t^0}$ and let $\widehat{t^1} = \sum_{T_i^1 \in T^1} t_i^1 / |T^1|$. As ε estimates the probability of a collection of random tests to expose an error, $1/\varepsilon$ serves as an estimation for the average number of tests required to expose that error. Since execution is stopped as soon as a test hits the error, the time to expose that error corresponds, on average, to the execution of a sequence of $\lceil 1/\varepsilon \rceil$ tests in which the first $\lceil 1/\varepsilon \rceil - 1$ tests do not expose the error, but the last one does, i.e., $(\lceil 1/\varepsilon \rceil - 1)\widehat{t^0} + t_j^1$. As there are exactly $|T^1|$ such sequences, by taking the arithmetic mean over them, we obtain the average effort in scenario $v = (p, n, s)$

$$EF(v) = \begin{cases} (\lceil 1/\varepsilon(v) \rceil - 1)\ \widehat{t^0(v)} + \widehat{t^1(v)} & \text{if } \varepsilon(v) \neq 0 \\ |T(v)|\ \widehat{t^0(v)} & \text{if } \varepsilon(v) = 0. \end{cases} \tag{1}$$

Given two generators G and g, we determined the relative effort of G with respect to g as the ratio $EF_g(v)/EF_G(v)$. We obtained the average improvement for designs containing a same type of error by taking the geometric mean over the set of all v. Similarly, when both generators expose the error in the same scenario v, we determine the relative effectiveness of G with respect to g as the ratio $\varepsilon_G(v)/\varepsilon_g(v)$.

We measured the *functional coverage* as the fraction of transitions covered in the state machine of each cache controller. For a design containing no errors, we tracked all memory blocks referenced by the collection of random tests induced by a given scenario $v = (p, n, s)$. We counted the number of *different* transitions taken in the machine tracking the state of the block corresponding to a given location a from the perspective of the cache owned by a given processor i at level L, written $\mathrm{TRAN}_a^{i,L}(v)$. Then we computed the transition coverage, written $TC_a^{i,L}(v) = \mathrm{TRAN}_a^{i,L}(v)/\mathrm{total}(L)$, where $\mathrm{total}(L)$ is the number of transitions of the state machine at level L. Next, we obtained the distribution of the transition coverage induced by v at level L, written $TC(v, L)$, i.e., the distribution of $TC_a^{i,L}(v)$ over all processors (i) and all locations (a). We also determined a similar distribution at the (shared) last-level cache, written $TC(v, L2)$. Finally, for each level L, we took the *median* over the collection of all scenarios v, written $\widehat{TC}(L)$. Given two generators G and g, we determined the relative coverage of G with respect to g as the ratio $\widehat{TC_G}(L)/\widehat{TC_g}(L)$.

The next sections report distinct evaluation approaches: the former shows the *relative* improvement with respect to the baseline over the whole generation space; the latter, the *absolute* values for designs with a fixed core count.

C. Broad Assessment of Impact

Table III reports the relative coverage at all levels. It indicates that pure chaining does not improve the typical coverage. However, it shows that pure biasing improves the coverage most significantly at lower hierarchical levels. The fact that the improvement increases from the highest to the lowest level shows how ineffective random address assignment is in face of the progressively larger associativities toward the lowest level. When combined, the techniques led to the highest improvement at all levels and every core count. Biasing was the largest contributor to the combined improvement at L2; chaining, the largest contributor at L0. This is a first evidence of the complementary nature of the proposed techniques, as it will be explained in Section VII-D.

The significant improvement in coverage observed for the combination of the proposed techniques is a general evidence of higher chances of exposing design errors in less time. To provide further support to that claim, we measured the potential, the effectiveness, and the effort to expose actual errors over a collection of faulty designs, as follows.

TABLE IV
FRACTIONS OF GENERATION SPACE WITH POTENTIAL FOR ERROR EXPOSURE (P−, C−, P+, AND C+ ARE ACRONYMS FOR PLAIN−, CHAIN−, PLAIN+, AND CHAIN+)

	Chaining										
	Exposure	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
8 cores	P− . C−	0.55	0.40	0.25	0.50	0.00	0.00	0.00	0.00	0.15	0.15
	P− . $\overline{\mathrm{C}}$−	0.05	0.00	0.00	0.10	0.00	0.00	0.00	0.00	0.05	0.10
	$\overline{\mathrm{P}}$− . C−	0.05	0.25	0.25	0.05	0.00	0.00	0.00	0.00	0.30	0.05
	$\overline{\mathrm{P}}$− . $\overline{\mathrm{C}}$−	0.35	0.35	0.50	0.35	1.00	1.00	1.00	1.00	0.50	0.70
	PLAIN−	0.60	0.40	0.25	0.60	0.00	0.00	0.00	0.00	0.20	0.25
	CHAIN−	0.60	0.65	0.50	0.55	0.00	0.00	0.00	0.00	0.45	0.20
16 cores	P− . C−	0.65	0.40	0.30	0.55	0.00	0.00	0.00	0.00	0.15	0.15
	P− . $\overline{\mathrm{C}}$−	0.05	0.05	0.10	0.05	0.00	0.00	0.00	0.00	0.05	0.20
	$\overline{\mathrm{P}}$− . C−	0.05	0.10	0.15	0.10	0.00	0.00	0.00	0.00	0.35	0.10
	$\overline{\mathrm{P}}$− . $\overline{\mathrm{C}}$−	0.25	0.45	0.45	0.30	1.00	1.00	1.00	1.00	0.45	0.55
	PLAIN−	0.70	0.45	0.40	0.60	0.00	0.00	0.00	0.00	0.20	0.35
	CHAIN−	0.70	0.50	0.45	0.65	0.00	0.00	0.00	0.00	0.50	0.25
32 cores	P− . C−	0.60	0.45	0.35	0.35	0.00	0.00	0.00	0.00	0.00	0.10
	P− . $\overline{\mathrm{C}}$−	0.20	0.20	0.20	0.25	0.00	0.00	0.00	0.00	0.05	0.10
	$\overline{\mathrm{P}}$− . C−	0.05	0.10	0.20	0.15	0.00	0.00	0.00	0.00	0.45	0.10
	$\overline{\mathrm{P}}$− . $\overline{\mathrm{C}}$−	0.15	0.25	0.25	0.25	1.00	1.00	1.00	1.00	0.50	0.70
	PLAIN−	0.80	0.65	0.55	0.60	0.00	0.00	0.00	0.00	0.05	0.20
	CHAIN−	0.65	0.55	0.55	0.50	0.00	0.00	0.00	0.00	0.45	0.20

	Biasing										
	Exposure	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
8 cores	P− . P+	0.60	0.40	0.15	0.35	0.00	0.00	0.00	0.00	0.20	0.20
	P− . $\overline{\mathrm{P}}$+	0.00	0.00	0.10	0.25	0.00	0.00	0.00	0.00	0.00	0.05
	$\overline{\mathrm{P}}$− . P+	0.40	0.60	0.65	0.40	0.60	0.65	0.50	0.60	0.80	0.65
	$\overline{\mathrm{P}}$− . $\overline{\mathrm{P}}$+	0.00	0.00	0.10	0.00	0.40	0.35	0.50	0.40	0.00	0.10
	PLAIN−	0.60	0.40	0.25	0.60	0.00	0.00	0.00	0.00	0.20	0.25
	PLAIN+	1.00	1.00	0.80	0.75	0.60	0.65	0.50	0.60	1.00	0.85
16 cores	P− . P+	0.70	0.45	0.25	0.45	0.00	0.00	0.00	0.00	0.20	0.25
	P− . $\overline{\mathrm{P}}$+	0.00	0.00	0.15	0.15	0.00	0.00	0.00	0.00	0.00	0.10
	$\overline{\mathrm{P}}$− . P+	0.30	0.50	0.35	0.40	0.50	0.75	0.50	0.70	0.80	0.55
	$\overline{\mathrm{P}}$− . $\overline{\mathrm{P}}$+	0.00	0.05	0.25	0.00	0.50	0.25	0.50	0.30	0.00	0.10
	PLAIN−	0.70	0.45	0.40	0.60	0.00	0.00	0.00	0.00	0.20	0.35
	PLAIN+	1.00	0.95	0.60	0.85	0.50	0.75	0.50	0.70	1.00	0.80
32 cores	P− . P+	0.80	0.65	0.30	0.35	0.00	0.00	0.00	0.00	0.05	0.15
	P− . $\overline{\mathrm{P}}$+	0.00	0.00	0.25	0.25	0.00	0.00	0.00	0.00	0.00	0.05
	$\overline{\mathrm{P}}$− . P+	0.20	0.35	0.15	0.40	0.50	0.75	0.50	0.60	0.95	0.70
	$\overline{\mathrm{P}}$− . $\overline{\mathrm{P}}$+	0.00	0.00	0.30	0.00	0.50	0.25	0.50	0.40	0.00	0.10
	PLAIN−	0.80	0.65	0.55	0.60	0.00	0.00	0.00	0.00	0.05	0.20
	PLAIN+	1.00	1.00	0.45	0.75	0.50	0.75	0.50	0.60	1.00	0.85

	Chaining and Biasing										
	Exposure	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
8 cores	P− . C+	0.60	0.40	0.25	0.60	0.00	0.00	0.00	0.00	0.20	0.25
	P− . $\overline{\mathrm{C}}$+	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	$\overline{\mathrm{P}}$− . C+	0.40	0.60	0.75	0.40	0.65	0.70	0.60	0.70	0.80	0.65
	$\overline{\mathrm{P}}$− . $\overline{\mathrm{C}}$+	0.00	0.00	0.00	0.00	0.35	0.30	0.40	0.30	0.00	0.10
	PLAIN−	0.60	0.40	0.25	0.60	0.00	0.00	0.00	0.00	0.20	0.25
	CHAIN+	1.00	1.00	1.00	1.00	0.65	0.70	0.60	0.70	1.00	0.90
16 cores	P− . C+	0.70	0.45	0.40	0.60	0.00	0.00	0.00	0.00	0.20	0.35
	P− . $\overline{\mathrm{C}}$+	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	$\overline{\mathrm{P}}$− . C+	0.30	0.55	0.60	0.40	0.60	0.75	0.55	0.70	0.80	0.55
	$\overline{\mathrm{P}}$− . $\overline{\mathrm{C}}$+	0.00	0.00	0.00	0.00	0.40	0.25	0.45	0.30	0.00	0.10
	PLAIN−	0.70	0.45	0.40	0.60	0.00	0.00	0.00	0.00	0.20	0.35
	CHAIN+	1.00	1.00	1.00	1.00	0.60	0.75	0.55	0.70	1.00	0.90
32 cores	P− . C+	0.80	0.65	0.55	0.60	0.00	0.00	0.00	0.00	0.05	0.20
	P− . $\overline{\mathrm{C}}$+	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	$\overline{\mathrm{P}}$− . C+	0.20	0.35	0.45	0.40	0.50	0.75	0.50	0.60	0.95	0.75
	$\overline{\mathrm{P}}$− . $\overline{\mathrm{C}}$+	0.00	0.00	0.00	0.00	0.50	0.25	0.50	0.40	0.00	0.05
	PLAIN−	0.80	0.65	0.55	0.60	0.00	0.00	0.00	0.00	0.05	0.20
	CHAIN+	1.00	1.00	1.00	1.00	0.50	0.75	0.50	0.60	1.00	0.95

TABLE V
AVERAGE IMPROVEMENT IN EFFECTIVENESS (UNDER JOINT EXPOSURE SUBSPACES) AND IN EFFORT (OVER THE ENTIRE GENERATION SPACE)

Metric	**p**	**Chaining** **CHAIN-** w.r.t **PLAIN-**										**Biasing** **PLAIN+** w.r.t **PLAIN-**										**Chaining and Biasing** **CHAIN+** w.r.t **PLAIN-**									
		D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
$\varepsilon_G/\varepsilon_g$	**8**	1.3	1.1	1.1	1.0	—	—	—	—	1.3	1.0	5.9	10.3	1.7	4.3	—	—	—	—	16.1	18.1	21.9	22.8	18.0	13.5	—	—	—	—	55.0	16.7
	16	1.5	1.1	1.0	1.5	—	—	—	—	1.3	0.6	7.3	10.3	1.0	2.3	—	—	—	—	10.1	7.8	25.7	18.2	13.3	10.1	—	—	—	—	41.8	4.4
	32	1.0	1.4	1.2	1.1	—	—	—	—	—	0.4	2.4	9.6	0.8	1.2	—	—	—	—	2.0	12.1	15.5	19.5	9.8	5.7	—	—	—	—	23.0	2.7
EF_g/EF_G	**8**	1.2	1.1	1.1	1.1	1.0	1.0	1.0	1.0	1.3	1.0	6.7	11.8	1.9	2.5	3.9	7.1	1.5	6.0	7.8	12.5	24.9	29.2	21.1	15.8	8.6	11.7	2.5	6.7	29.7	14.9
	16	1.3	1.0	1.0	1.4	1.0	1.0	1.0	1.0	1.2	0.8	4.6	7.3	0.7	1.9	4.0	10.5	1.4	5.3	5.7	8.1	20.4	17.4	10.6	9.4	7.8	15.4	1.9	6.1	26.7	7.5
	32	1.0	1.2	1.1	1.0	1.0	1.0	1.0	1.0	1.2	1.0	2.1	6.4	0.6	1.0	2.8	12.0	1.8	5.9	3.2	9.8	12.2	12.2	8.5	6.7	6.0	15.4	1.9	5.7	19.4	6.7

Table IV shows the fractions of verification scenarios with potential for error exposure. Note that a value in that table represents an optimistic estimate for the actual effectiveness. For instance, an entry containing a unit value does not mean that all tests are able to expose a given type of error. Instead, it means that all *verification scenarios* have potential of exposing it, but the actual detection is largely independent of the (p, n, s) parameters: either it may depend on the probabilistic parameters (seed and mix) or on the specific features of each generator. For each core count, the table has two segments: one reporting the correlation of potential exposure for pairs of generators, another showing the overall exposure of each generator individually.

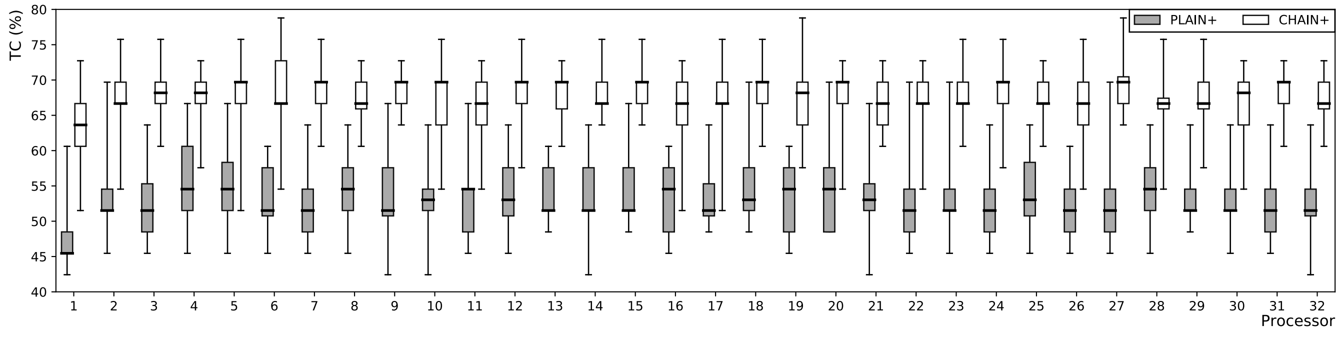
For each core count, the first row captures the fraction of the generation space where the features common to the baseline and the proposed generators were sufficient for error exposure. Such common features narrow down to the random choice of operations and addresses, as well as the exploitation of conventional constraints (e.g., target operation mix). The results indicate that the role of such features vary among designs: it might be significant for some (D0–D3, D8, and D9), but negligible for others (D4–D7).

The second row shows the fraction of the generation space where the baseline exposes an error that is not exposed by one of the proposed generators. Under pure chaining, that fraction was not negligible for a few designs (32-core D0–D3 and 16-core D9). Under pure biasing, that fraction was zero or negligible for all designs but three (32-core, D2 and D3, and 8-core D3). When the techniques were combined, that fraction was zero for all designs. Thus, the baseline generator is rarely superior to a *biased* generator, especially when it also exploits chaining.

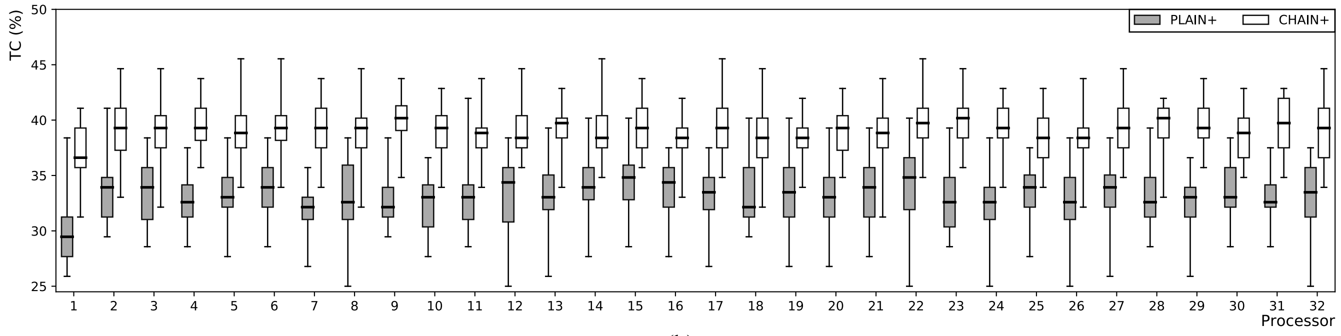
The third row shows the fraction of the generation space in which one of the proposed generators exposed an error that was not exposed by the baseline. Under pure chaining that fraction was significant for a few designs (D1, D2, and D8). In contrast, biasing significantly increased that fraction (whether pure or combined). The combination of biasing and chaining was superior over at least 20% and at most 95% of the generation space, as compared to the baseline. Thus, in general, a biased generator is superior for a significant fraction of the generation space, especially when it also exploits chaining.

The fourth row shows the fraction of the space where neither the features common to all generators nor the proposed complementary features were able to foster exposure. Biasing significantly reduced that fraction as compared to pure chaining, except for one design (32-core D2). The combination of biasing and chaining completely ruled out the joint non-exposure subspace for half of the designs (D0–D3 and D8). The other half (D4–D7 and D9) gives evidence of challenging errors that can only be exposed if extra constraints (such as the proposed ones) prune random test generation.

The last two rows show the overall impact of each generator. Pure biasing increased the fraction of the space leading to the exposure of errors in all designs but one (32-core D2). This indicates that biasing, in general, makes error detection less dependent on the choice of parameters, i.e., it makes constrained random choice useful over a larger fraction of the generation space. Indeed, within the ranges adopted for the parameters, pure biasing made detection practically independent of the choice of n and s for three designs (D0, D1, and D8), regardless of core count. When biasing was combined with chaining, the fraction leading to exposure increased in *all* designs as compared to the baseline, and at least 50% of



(a)



(b)

Fig. 7. Impact on the functional coverage of private cache controllers for 32-core designs (for tests with 4K operations and 32 locations). (a) Transition coverage distribution at level L0. (b) Transition coverage distribution at level L1.

all scenarios became suitable for exposing errors. The combination made detection practically independent of the choice of *n* and *s* for six designs (D0, D1, D2, D3, D8, and D9) instead of three. This shows one of the contributions of chaining: it may help in extending the potential for error exposure over a larger number of scenarios. Another contribution of chaining (which cannot be inferred from Table IV) comes into play when a given scenario is selected for a test. Chaining tends to increase the probability of error detection, making constrained random choice more effective and efficient, as shown next.

The top of Table V reports the relative effectiveness over joint exposure subspaces. For four designs (D4–D7), the baseline generator was not able to expose errors with test lengths between 1K and 16K (empty subspaces are indicated by dashes). Pure chaining kept or slightly improved the effectiveness for all designs but two (16/32-core D9). In contrast, pure biasing significantly improved the effectiveness for all designs but two (16/32-core D2). Finally, the combination of biasing and chaining improved the effectiveness for all designs. Note that the combination of the techniques led to the largest improvements with respect to the baseline for all designs but three (8/16/32-core D9). This can be explained by the fact that the structure of canonical multiprocessor chains foster operation conflict, which is the main mechanism required to expose errors in most designs.

When two generators are compared in *identical* verification scenarios where *both* expose an error, the improvement in effectiveness is directly reflected as an improvement in effort. However, since test suites usually exploit distinct settings of parameters, the impact on effort over the entire generation space provides a better assessment for test throughput. That is, why we relaxed the joint exposure requirement to evaluate the impact on effort, as follows.

The bottom of Table V shows the relative effort over the entire generation space, whether both, one, or none of the generators under comparison happened to expose a given type of error. Pure chaining hardly improved the effort. In contrast, pure biasing significantly improved the effort for all but three designs (32-core D3 and 16/32-core D2). The combination of biasing and chaining improved the effort for all designs. As compared to pure biasing, the combination led to large reductions in effort, which resulted from the gains in effectiveness under joint exposure (Table V) and from the higher numbers of scenarios leading to error exposure (Table IV). Albeit the combination worsened the effort for three designs (32-core D7 and 16/32-core D9), the maximum degradation (32%) was much smaller than the maximum improvement (15 times).

The experimental evidence indicates that pure chaining does not pay off. However, it indicates that address biasing not only pays off, but it also favors proper operation chaining, which tends to further reduce the verification effort.

D. Assessment for Fixed Core Count

Fig. 7 displays the coverage distributions resulting from the execution of all random tests induced by a given scenario from the perspective of each private cache. Since it is clear from Section VII-C that only biased generators can significantly improve coverage, Fig. 7 omits the plots for PLAIN− and CHAIN−, whose typical transition coverage was around 19%–21% at L1 and 45%–49% at L0. As compared to those values, Fig. 7 shows that biasing significantly improved

TABLE VI
IMPACT ON EFFORT (AND ERROR EXPOSURE) FOR 32-CORE DESIGNS CONTAINING SWMR VIOLATIONS (WHEN EXPLOITING 32 SHARED LOCATIONS)

	Unbiased								**Biased**							
	PLAIN-				**CHAIN-**				**PLAIN+**				**CHAIN+**			
n	**D0**	**D1**	**D2**	**D3**	**D0**	**D1**	**D2**	**D3**	**D0**	**D1**	**D2**	**D3**	**D0**	**D1**	**D2**	**D3**
1k	177 (0.07)	702 (0.02)	701 (0.02)	237 (0.05)	232 (0.05)	685 (0.02)	685 (0.02)	349 (0.03)	92 (0.18)	25 (0.63)	966 (0.02)	986 (0.00)	**25** (0.82)	**25** (0.78)	**70** (0.20)	**229** (0.07)
2k	387 (0.03)	759 (0.02)	763 (0.00)	776 (0.02)	754 (0.02)	182 (0.07)	737 (0.02)	755 (0.00)	94 (0.22)	30 (0.82)	1324 (0.02)	1252 (0.00)	**29** (0.95)	**28** (0.92)	**67** (0.27)	**225** (0.08)
4k	81 (0.17)	135 (0.10)	163 (0.08)	138 (0.10)	133 (0.10)	197 (0.07)	264 (0.05)	269 (0.05)	91 (0.25)	35 (0.77)	1588 (0.00)	1610 (0.00)	**36** (0.92)	**35** (0.92)	**61** (0.43)	**191** (0.13)
8k	236 (0.07)	472 (0.03)	947 (0.02)	317 (0.05)	182 (0.08)	302 (0.05)	302 (0.05)	**308** (0.05)	86 (0.33)	42 (0.87)	2039 (0.02)	2072 (0.00)	**10** (1.00)	**42** (0.93)	**47** (0.57)	342 (0.10)
16k	129 (0.15)	166 (0.12)	224 (0.08)	287 (0.07)	90 (0.23)	129 (0.15)	168 (0.12)	**169** (0.12)	118 (0.33)	61 (0.77)	1463 (0.03)	3031 (0.00)	**55** (0.97)	**57** (0.97)	**66** (0.73)	285 (0.18)

TABLE VII
IMPACT ON EFFORT (AND ERROR EXPOSURE) FOR 32-CORE DESIGNS CONTAINING DV VIOLATIONS (WHEN EXPLOITING 32 SHARED LOCATIONS)

	Unbiased												**Biased**											
	PLAIN-						**CHAIN-**						**PLAIN+**						**CHAIN+**					
n	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**
1k	658 (0.00)	645 (0.00)	720 (0.00)	710 (0.00)	645 (0.00)	658 (0.00)	642 (0.00)	628 (0.00)	703 (0.00)	693 (0.00)	312 (0.03)	641 (0.00)	56 (0.28)	**8** (1.00)	**202** (0.08)	**27** (0.53)	131 (0.12)	926 (0.00)	**23** (0.97)	**8** (1.00)	485 (0.03)	73 (0.23)	**23** (0.88)	876 (0.00)
2k	718 (0.00)	705 (0.00)	780 (0.00)	769 (0.00)	704 (0.00)	717 (0.00)	694 (0.00)	680 (0.00)	756 (0.00)	745 (0.00)	681 (0.00)	694 (0.00)	50 (0.42)	**8** (1.00)	**255** (0.08)	32 (0.73)	89 (0.23)	**290** (0.07)	**9** (1.00)	**8** (1.00)	1246 (0.00)	**32** (0.65)	**27** (0.92)	1119 (0.02)
4k	790 (0.00)	778 (0.00)	855 (0.00)	843 (0.00)	778 (0.00)	154 (0.08)	758 (0.00)	746 (0.00)	821 (0.00)	810 (0.00)	741 (0.02)	756 (0.02)	63 (0.45)	**8** (1.00)	210 (0.13)	**39** (0.85)	86 (0.28)	1541 (0.00)	**34** (0.97)	**8** (1.00)	**122** (0.20)	42 (0.83)	**33** (0.87)	**1510** (0.02)
8k	907 (0.00)	897 (0.00)	975 (0.00)	960 (0.00)	895 (0.00)	908 (0.00)	876 (0.00)	869 (0.00)	942 (0.00)	929 (0.00)	859 (0.02)	876 (0.00)	46 (0.53)	**8** (1.00)	525 (0.07)	49 (0.97)	214 (0.15)	2010 (0.00)	**9** (1.00)	**8** (1.00)	**368** (0.10)	**48** (0.97)	**42** (0.97)	**518** (0.07)
16k	1116 (0.00)	1106 (0.00)	1191 (0.00)	1172 (0.00)	1106 (0.00)	1109 (0.02)	1128 (0.00)	1116 (0.00)	1200 (0.00)	1182 (0.00)	212 (0.08)	1127 (0.02)	64 (0.63)	**8** (1.00)	1052 (0.05)	**12** (1.00)	162 (0.30)	**1451** (0.03)	**9** (1.00)	**8** (1.00)	**267** (0.20)	16 (1.00)	**53** (0.97)	3229 (0.02)

coverage whether alone (PLAIN+) or combined with chaining (CHAIN+). Indeed, the combination led to the highest coverage for all private caches at the same level.

The higher coverage of the combination of chaining with biasing can be explained as follows. Chaining and biasing foster, respectively, conflict and eviction events. Such different classes of events often induce *distinct* transitions in the state machine controlling a private cache, which contributes to raising the coverage. At all levels, biasing fosters eviction-induced transitions. At L0, biasing also fosters transitions from state I, whereas chaining fosters transitions from states E, S, and M either through intraprocessor conflict [i.e., (S, M) and (E, M)] or through interprocessor conflict [i.e., (M, I), (E, I), and (S, I)]. At L1, biasing fosters transitions induced by GETS and GETX requests from L0, while chaining fosters transitions induced not only by UPG requests from L0, but also requests from other cores (Invalidate, FWD_GETS, FWD_GETX). Note that the impact of combining biasing and chaining is higher at L0 than at L1. One reason for that results from the exploitation of inclusive caches: operation conflicts raising distinct types of requests (Invalidate, UPG, FWD_GETS, FWD_GETX) often trigger the *same* transition at L1. Another reason is the smaller frequency of evictions due to the higher associativity at L1.

At L2 (where the shared cache also plays the role of directory), the median coverage was 15% for both PLAIN− and CHAIN− and 53% for *both* PLAIN+ and CHAIN+. Chaining had marginal impact, because most requests resulting from interprocessor conflict (Invalidate, FWD_GETS, FWD_GETX) do not induce transitions at L2 (they are actually outputs actions). Only 4 out 148 transitions are induced by intra (UPG) or interprocessor conflict (GETX, GETS). Thus, most of the transitions at L2 are fostered by biasing.

Tables VI and VII convey three complementary pieces of information. Each table shows whether or not a generator was able to expose an error with a given test length when targeting 32-core designs (an entry filled in black indicates that the error was undetected). It also shows the required effort (expressed in seconds) to expose an error or the effort wasted in trying to expose it. Values in bold indicate the generator leading to the minimum effort for each verification scenario that exposed a given type of error. Finally, it shows the effectiveness of each generator to expose a given type of error in every scenario (as indicated between parentheses).

Let us first focus on random tests with 1K operations. In Table VII, note that no unbiased generator ever exposed errors for designs D4–D7 and D9, but biased generators exposed all errors except for D9. Table VI shows that the probability of a random test to expose the error in design D2 is 1/50 for PLAIN+ and 1/5 for CHAIN+, i.e., PLAIN+ requires 50 tests on average to expose the error, while CHAIN+ requires only 5. This explains why CHAIN+ requires one order of magnitude less effort to expose the error in that design.

Let us now analyze the impact over the range of test lengths from 1K to 16K. For errors leading to SWMR violations (Table VI), neither pure chaining nor pure biasing significantly improved the effort. However, their combination led to the smallest effort for almost all designs and test lengths. Errors leading to DV violations (Table VII) were much harder to find. Pure chaining improved exposure for a single design (D8), whereas biasing significantly improved error detection for all designs. Their combination exposed the errors in every design for all cases but two. Although CHAIN+ found practically as many errors as PLAIN+, the former required the smallest effort in most cases.

Finally, we measured the time to *generate* tests for 32-core designs with 16*k* operations and 32 locations. On average, PLAIN− and PLAIN+ took 0.3 s, while CHAIN− and CHAIN+ took 0.8 s (i.e., one to four orders of magnitude smaller than the *verification* efforts in Tables VI and VII).

VIII. Conclusion and Perspectives

The experimental results indicate that the proposed techniques are complementary and improve constrained random test generation when a reusable memory-model checker is exploited for verification. Address biasing makes error detection less dependent on generation parameters, while operation chaining raises the probability of error detection. We have shown design cases for which the combination of biasing and chaining largely reduced the effort. In half of the 32-core designs, the baseline generator was unable to expose the same errors detected by our techniques with tests 16 times shorter. Albeit we had to limit the analysis to designs containing ten types of errors, the observed improvement in coverage indicates that the adequacy of the techniques is not limited to the evaluated design cases.

We plan to relax the current uniform-thread-length constraint, and to exploit the techniques for building novel directed-test generators targeting presilicon verification.

Acknowledgment

The authors would like to thank E. Rambo, L. Freitas, and O. Henschel for the legacy infrastructure enabling this paper.

References

[1] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Commun. ACM*, vol. 55, no. 7, pp. 78–89, Jun. 2012.

[2] S. Devadas, "Toward a coherent multicore memory model," *Computer*, vol. 46, no. 10, pp. 30–31, Oct. 2013.

[3] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[4] D. Abts, S. Scott, and D. J. Lilja, "So many states, so little time: Verifying memory coherence in the Cray X1," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2003, p. 10.

[5] M. Zhang, J. D. Bingham, J. Erickson, and D. J. Sorin, "PVCoherence: Designing flat coherence protocols for scalable verification," *IEEE Micro*, vol. 35, no. 3, pp. 84–91, May/Jun. 2015.

[6] A. Adir and G. Shurek, "Generating concurrent test-programs with collisions for multi-processor verification," in *Proc. 7th IEEE Int. High Level Design Validation Test Workshop*, 2002, pp. 77–82.

[7] A. Adir *et al.*, "Genesys-pro: Innovations in test program generation for functional processor verification," *IEEE Design Test Comput.*, vol. 21, no. 2, pp. 84–93, Mar. 2004.

[8] Y. Naveh *et al.*, "Constraint-based random stimuli generation for hardware verification," *AI Mag.*, vol. 28, no. 3, p. 13, 2007.

[9] I. Wagner and V. Bertacco, "MCjammer: Adaptive verification for multicore designs," in *Proc. Design Autom. Test Europe (DATE)*, 2008, pp. 670–675.

[10] X. Qin and P. Mishra, "Automated generation of directed tests for transition coverage in cache coherence protocols," in *Proc. Design Autom. Test Europe (DATE)*, 2012, pp. 3–8.

[11] O. Shacham *et al.*, "Verification of chip multiprocessor memory systems using a relaxed scoreboard," in *Proc. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2008, pp. 294–305.

[12] L. S. Freitas, E. A. Rambo, and L. C. V. dos Santos, "On-the-fly verification of memory consistency with concurrent relaxed scoreboards," in *Proc. Design Autom. Test Europe (DATE)*, 2013, pp. 631–636.

[13] M. Elver and V. Nagarajan, "McVerSi: A test generation framework for fast memory consistency verification in simulation," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2016, pp. 618–630.

[14] K. Gharachorloo, "Memory consistency models for shared-memory multiprocessors," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Stanford Univ., Stanford, CA, USA, 1995.

[15] S. Hangal, D. Vahia, C. Manovit, J.-Y. J. Lu, and S. Narayanan, "TSOtool: A program for verifying memory systems using the memory consistency model," *ACM SIGARCH Comput. Archit. News*, vol. 32, no. 2, pp. 114–123, Mar. 2004.

[16] G. A. G. Andrade, M. Graf, and L. C. V. dos Santos, "Chain-based pseudorandom tests for pre-silicon verification of CMP memory systems," in *Proc. 34th IEEE Int. Conf. Comput. Design (ICCD)*, 2016, pp. 552–559.

[17] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models," in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds. Berlin, Germany: Springer, 2010, pp. 258–272. [Online]. Available: https://link.springer.com/chapter/10.1007%2F978-3-642-14295-6_25

[18] J. Alglave *et al.*, "GPU concurrency: Weak behaviours and programming assumptions," in *Proc. 20th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2015, pp. 577–591.

[19] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux, "Automated synthesis of comprehensive memory model litmus test suites," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2017, pp. 661–675.

[20] C. Manovit and S. Hangal, "Completely verifying memory consistency of test program executions," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2006, pp. 166–175.

[21] A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang, *Fast and Generalized Polynomial Time Memory Consistency Verification* (LNCS 4144). Heidelberg, Germany: Springer, 2006, pp. 503–516. [Online]. Available: https://link.springer.com/chapter/10.1007/11817963_46

[22] Y. Chen *et al.*, "Fast complete memory consistency verification," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2009, pp. 381–392.

[23] W. Hu, Y. Chen, T. Chen, C. Qian, and L. Li, "Linear time memory consistency verification," *IEEE Trans. Comput.*, vol. 61, no. 4, pp. 502–516, Apr. 2012.

[24] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[25] E. A. Rambo, O. P. Henschel, and L. C. V. dos Santos, "Automatic generation of memory consistency tests for chip multiprocessing," in *Proc. IEEE Int. Conf. Electron. Circuits Syst. (ICECS)*, 2011, pp. 542–545.

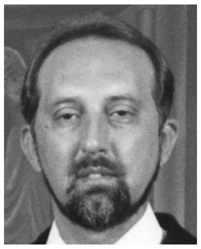
[26] E. A. Rambo, O. P. Henschel, and L. C. V. dos Santos, "On ESL verification of memory consistency for system-on-chip multiprocessing," in *Proc. Design Autom. Test Europe (DATE)*, 2012, pp. 9–14.

Gabriel A. G. Andrade received the B.Sc. and M.Sc. degrees in computer science from the Federal University of Santa Catarina, Florianópolis, Brazil, in 2014 and 2016, respectively, where he is currently pursuing the Ph.D. degree in automation and systems engineering.

His current research interests include algorithms for test generators and runtime checkers targeting multicore chip verification.

Marleson Graf received the B.Sc. degree in computer science from the Federal University of Santa Catarina, Florianópolis, Brazil, in 2017, where he is currently pursuing the M.Sc. degree in computer science.

His current research interests include algorithms for test generators and runtime checkers targeting multicore chip verification.



Luiz C. V. dos Santos received the Doctoral degree from the Eindhoven University of Technology, Eindhoven, The Netherlands, in 1998.

He was with the Design Automation Section of the Information and Communication Systems Group, Eindhoven University of Technology. He is a Professor with the Federal University of Santa Catarina, Florianópolis, Brazil. His current research interests include algorithms for verification and test of multicore chips.