Spec&Check: An Approach to the Building of Shared-Memory Runtime Checkers for Multicore Chip Design Verification

Marleson Graf, Olav P. Henschel, Rafael P. Alevato, Luiz C.V. dos Santos
Federal University of Santa Catarina
Florianpolis, SC, Brazil
(marleson.graf, olav.henschel)@posgrad.ufsc.br, rafael.alevato@grad.ufsc.br, luiz.santos@ufsc.br

***Abstract*—Multicore architectures are likely to largely relax sequential consistency constraints on store atomicity and on the ordering between loads and stores, while preserving a coherent shared-memory abstraction. As a result, multicore chip design verification is challenged by the higher number of valid execution witnesses resulting from consistency relaxation and by the larger coherence protocol's state space induced by growing core counts. On the one hand, litmus test generation is effective in exposing consistency bugs, but their coverage of coherence events is limited. On the other hand, random test generation (RTG) leads to higher coverage, but requires specialized checkers and higher observability for detecting subtle consistency errors. Unfortunately, under RTG, no reported checker is able to handle the non-multiple-copy atomic (nMCA) stores arising from full relaxation. This paper proposes an approach that bridges that gap. It relies on an nMCA-compliant abstract specification for shared memory behavior and on an observability template for guiding the insertion of proper monitors into the design representation. We compared a checker built under the novel approach with a conventional one, when running the same test suites, each built with many programs of fixed size. For 4K-instruction programs, the conventional checker raised false positives for 1/3 of the test suites when targeting correct 32-core nMCA designs, whereas the new checker raised none. The improved verification quality resulting from the more general specification of memory behavior came at the expense of negligible overhead, and often led to effort reduction.**

I. Introduction

On-chip hardware coherence can scale gracefully as the number of cores grows [1]. Coherent memory models can be expected even in the scale of a few hundreds of cores [2].

A memory model [3] defines shared-memory behavior for programmers and chip designers. It specifies the ordering of accesses to distinct locations (consistency rules), the ordering of stores to the same location (coherence requirement), and when a value written by a store can be observed by loads in the same or in other cores (store atomicity). Sequential Consistency (SC) [4] fully enforces program ordering on memory accesses, and leads to *multiple-copy atomic* (MCA) [5] stores. Most manufacturers have been building processors that relax SC. The x86 model allows a non-conflicting load to overtake a store that precedes it in program order, and it provides *read-own-write-early-multiple-copy atomic* (rMCA) stores. Modern architectures (e.g. IBM Power9, ARMv8, and RISC-V) largely relax program order, and allow *non-multiple-copy atomic* (nMCA) stores.

Such architectures challenge shared-memory verification, because their relaxation of order and store atomicity increases the number of valid execution witnesses as compared to SC, making it harder for functional verification to *expose* invalid ones. Besides, the relaxation of store atomicity makes it harder to *tell* a valid execution witness from an invalid one, because conventional checkers assume rMCA stores [6]–[9]. When targeting an *nMCA design*, a conventional rMCA checker is bound to raise false errors, because rMCA is a stronger condition than nMCA, whereas an nMCA checker is effective in detecting actual errors for a faulty design. However, when targeting an *rMCA design*, an nMCA checker may be less effective in raising actual error diagnoses as compared to an rMCA checker, because some invalid behavior under rMCA may be valid under nMCA. Thus, the building of a checker requires an approach able to handle either rMCA or nMCA implementations of a same architecture.

For these reasons, this paper proposes two main contributions:

1) An *abstract specification* for building checkers targeting relaxed memory models that comply with either rMCA or nMCA stores. It partitions memory behavior into architecture-dependent and architecture-independent axioms (the former being reusable; the latter, deserving target customization). In particular, we show how the latter are able to properly capture the complex behavior of cumulative fences.
2) An *observability template* that pinpoints where to insert monitors in the design representation such that the observed physical events can be used as proxies for the abstract events from our axiomatic specification. It is largely independent of micro-architecture, because it restricts monitors to structures common to most implementations: L1 cache buffers and reorder buffers (or similar structures for handling out-of-order execution). As a result, the dependence on design is narrowed down to monitor implementation, and it does not affect monitor location.

We relied on the proposed specification to build checkers for the ARMv8 architecture, and we used the proposed template for inserting monitors into representations of rMCA and nMCA designs.

This paper is organized as follows. Section II briefly reviews related work. Section III describes the axioms of our abstract specification. Section IV shows how the observability template maps abstract to physical events. Section V shows how a few properties and constraints can be exploited for building efficient checkers. Section VI evaluates checkers built under the proposed approach as compared to a conventional one. Section VII puts our conclusions in perspective.

II. Related Work

The automated generation of litmus tests have been proposed [10]–[12] for validating multicores. Despite its success in finding subtle bugs when testing commercial chips, this approach would have limited coverage at design time [13]. Besides, when a litmus test finds an error, it does not directly indicate where the error lies in the micro-architecture (i.e. no support is offered for design debugging).

Tricheck [14] is a full-stack approach for verifying whether the language, the compiler, the ISA, and the implementation collectively satisfy memory model requirements. It was able to uncover under-specifications and potential inefficiencies in the RISC-V ISA. It does handle nMCA stores, but its underlying micro-architecture checker is based on litmus tests, thereby limiting its coverage potential.

This work was partially supported by CNPq (GM 131114/2012-3, PIBIC 115271/2018-0) and CAPES/Brazil (Finance Code 001).

978-1-7281-2350-9/19/$31.00 ©2019 IEEE

Random test generation (RTG) and directed test generation (DTG) can lead to higher coverage than litmus tests. Several postmortem checkers were developed for analyzing memory traces resulting from running random tests on the prototype chip [6], [8]. However, if reused at design time, their limited observability (one monitor per core domain) would result in poor scalability with the growing number of cores and in limited support for error debugging.

In contrast, pre-silicon runtime checkers monitor multiple points per core domain [7], [9]. Due to the higher observability at design time, they are scalable, and their multiple monitors have the advantage of better locating where the error lies, which eases debugging. Unfortunately, reported runtime checkers assume rMCA stores. As a result, they eventually raise false error diagnoses for nMCA designs, which would uselessly increase debugging time.

Thus, to face modern architectures, growing core counts, coverage goals, and effort requirements at design time, a checker should:

1) handle nMCA stores (to rule out false error diagnoses),
2) rely on multiple monitors (to be scalable),
3) comply with RTG or DTG (to avoid limiting coverage),
4) stop simulation as soon as an error is hit (to reduce effort).

Sections III to V show how to build checkers with such features.

III. Abstract Specification

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

A. Abstract Events

We adopt the following granularity for the memory events: (1) each store is split into multiple copies, one for each processor, (2) each copy is further split into commit and completion phases. To capture pipeline effects on memory behavior, (3) each load is single-copy, and (4) each load is split into commit and completion phases. Gharachorloo [15] relied on (1) for proposing a formal rMCA-compliant specification, and on (2) and (3) for *informally* picturing how nMCA stores could be handled, but not on (4), since loads were considered atomic. In contrast to [15], we directly combine all the fine-grain events resulting from (1), (2), (3) and (4) into a single *formal* abstract specification that is suitable for functional verification.

Consider an *operation* O_j, issued by processor i, which makes reference to some location a, written $(O_j)^i_a$. Let L or S replace O to denote that the operation is either a load or a store. An operation gives rise to multiple *events*, but not all of them are relevant for specifying memory behavior. That is why *abstract* events are usually employed for specification [15], [16]. They represent the time when stores and loads take effect with respect to a given processor, i.e. they represent the ultimate effect of a chain of *physical* events throughout the memory hierarchy. We adopt the following abstract events.

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

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

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

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

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

An *execution witness* represents a distinct outcome of a parallel program, and it is determined by which store operation is observed by each load. An execution induces some memory *behavior*, which can be seen as a totally ordered set of memory *events* [15]. Every valid behavior must satisfy a partial order $\leq$ on the set of memory events. Valid memory behaviors are defined by ordering constraints, which are imposed by cache coherence, data dependencies, and to which extent program order is preserved by memory consistency.

It is well known that non-deterministic parallel programs tend to expose shared-memory bugs faster than real-life synchronized programs [6], [7], [13]. Therefore, test generation is often constrained to synthesize simple, non-deterministic programs stressing memory accesses. For simplicity, let us assume that the instruction sequences of a synthetic program: (1) do not contain locks, (2) may form data dependencies through memory, but not through registers (i.e. neither address nor source register dependencies are synthesized), and (3) do not form control dependencies (but are still suitable to speculation on address disambiguation). The next two sections formalize axioms that specify valid behaviors under such assumptions.

B. Architecture-independent behavior

To illustrate the axioms, Fig.1 pictures commit, completion, and availability events as white, black, and gray circles, respectively.

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

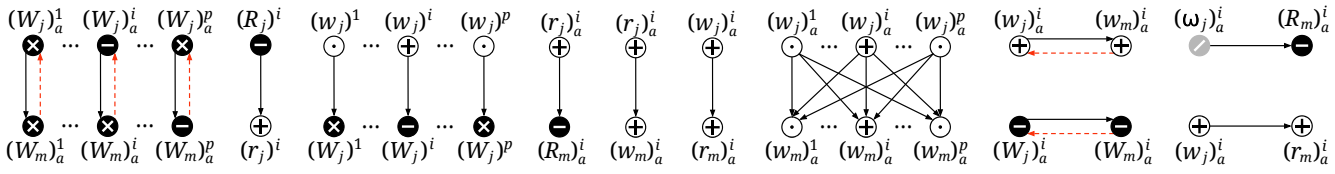


Fig. 1. Pictorial representations for Axioms 1 to 6 ($\oplus$ and $\ominus$ mark events at the issuing core; $\otimes$ and $\odot$, events induced at remote cores).

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

A load completes before it may be committed, a store commits before it completes with respect to every processor, as follows.

Axiom 2. ***Load completion-commit constraint*** *(Fig. 1b):*
$\forall L_j \in \mathcal{L} : R_j \leq r_j$

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

The order between operations to distinct locations has been fully relaxed in modern architectures, but program-order *commit* must be preserved at least for operations colliding at a same location. *Completion* order between *conflicting*[2] operations [15] can be relaxed as long as WAR, WAW, and RAW hazards[3] [17] are prevented in the scope of each thread (whether through memory or through read forwarding). Besides, program order must be preserved for loads colliding at a same location, because illegal outcomes would otherwise occur in lock-free programs if stores conflicting to the same location were observed by those loads [18], [19]. For proper ordering of colliding loads, the commit of the first one must occur before the completion of the second[4]. This is formalized below.

Axiom 4. ***Preserved program order constraints*** *(Fig. 1d-g):*
For all $O_j, O_m \in \mathcal{O}_a^i$, the following must hold in the case of

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

The last clause of Axiom 4 ensures that conflicting stores are committed in program order by waiting for the first store to commit with respect to every processor before the second is committed with respect to any processor [15]. Note that the combination of the second clause of Axiom 4 with Axioms 2 and 3, rules out WAR hazards through memory[5]. To prevent WAW hazards through memory, the actual updating of memory must occur in program order [17], i.e.:

Axiom 5. ***Local store completion order constraint*** *(Fig. 1h):*
$\forall S_j, S_m \in \mathcal{S}_a^i : (w_j)_a^i \leq (w_m)_a^i \Rightarrow (W_j)_a^i \leq (W_m)_a^i$

RAW hazards must be prevented through memory *and* read forwarding. Thus, for every conflicting store preceding a load in program order, the value to be written by the store must have been made available in a local buffer, before the actual reading takes place[6]:

Axiom 6. ***Store value availability constraint:*** *(Fig. 1i):*
$\forall S_j, L_m \in \mathcal{O}_a^i : (w_j)_a^i \leq (r_m)_a^i \Rightarrow (\omega_j)_a^i \leq (R_m)_a^i$

Now, let us define how a load L_a^i observes the value it returns. L_a^i may observe the value made available by some conflicting store that precedes the load in program order but has not completed with respect to processor i before the reading takes place, i.e.:

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

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

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

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

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

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

Let $Val[(O_j)_a^i]$ be the value written or returned by some operation issued by processor i. The value returned by L_a^i corresponds to the last[7] observed store, unless none was observed, when the initial value is returned, as follows.

Axiom 7. ***Returned value constraint:*** $\forall L \in \mathcal{L}_a^i$:

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

Axiom 7 is pictured in Section III-D, along with the axioms formalized in the next section.

C. Architecture-dependent behavior

Fences are sometimes used to restore program order between operations that are not data dependent (non-cumulative behavior),

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

[2] Two operations conflict if they collide at the same location and at least one is a store.

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

[4] As $(r_j)_a^i \leq (R_m)_a^i \wedge (R_m)_a^i \leq (r_m)_a^i \Rightarrow (r_j)_a^i \leq (r_m)_a^i$, Axiom 4, clause 1, combined with Axiom 2 results in a necessary and sufficient condition for preventing the so-called "RAR hazard" acknowledged as a bug in [19].

[5] $(R_j)_a^i \leq (r_j)_a^i \wedge (r_j)_a^i \leq (w_m)_a^i \wedge (w_m)_a^i \leq (W_m)_a^i \Rightarrow (R_j)_a^i \leq (W_m)_a^i$.

[6] This is a precondition for proper value consumption, whether it is obtained from memory or through read forwarding. The appropriate choice is specified by Axiom 7.

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

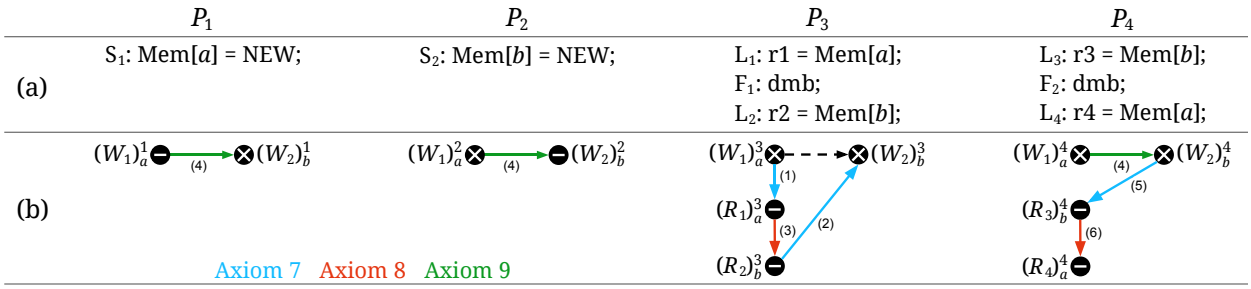


Fig. 2. Independent-read, independent-write (IRIW) example

and sometimes they also guarantee proper ordering in chains of memory operations spanning multiple threads (cumulative behavior). Our abstract specification encodes such complementary behaviors in distinct axioms, as follows.

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

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

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

In an nMCA micro-architecture, cumulative fences must be used when a thread is required to observe the same ordering of writes in other threads. The cumulative behavior stems from the following property: each store performed (or locally observed) before a fence must be ordered before each store performed (or locally observed) after the fence, and such order must be *globally* enforced in all cores observing those stores, as formalized below and illustrated next.

Definition 5. The set of stores *not* globally observable by L_a^i is $\overline{\sigma}_G(L_a^i) = \{S_j \in \mathcal{S}_a : R_a^i \leq (W_j)_a^i\}$.

Definition 6. The first store that is *not* globally observed by L_a^i, written $Min[\overline{\sigma}_G(L_a^i)]$, is the operation $S_j \in \overline{\sigma}_G(L_a^i)$ such that $\forall S_x \in \overline{\sigma}_G(L_a^i) : R_a^i \leq (W_j)_a^i \leq (W_x)_a^i$.

Axiom 9 captures the cumulative behavior of a fence F, which is similar to Power's `hwsync` and to ARM's `dmb isb`.

Axiom 9. ***Cumulative store order constraint:*** *Let $S_k = Max[\sigma_G(L_j)]$ and $S_n = Min[\overline{\sigma}_G(L_m)]$. For $O_j, O_m \in \mathcal{O}^i$, the following must hold in the case of*

- $L_j \prec_{po} F \prec_{po} L_m : \sigma_G(L_j) \neq \emptyset \wedge \sigma_G(L_m) = \emptyset \Rightarrow (W_k)^{x \neq i} \leq (W_n)^{x \neq i}$
- $L_j \prec_{po} F \prec_{po} S_m : \sigma_G(L_j) \neq \emptyset \Rightarrow (W_k)^{x \neq i} \leq (W_m)^{x \neq i}$
- $S_j \prec_{po} F \prec_{po} L_m : \sigma_G(L_m) = \emptyset \Rightarrow (W_j)^{x \neq i} \leq (W_n)^{x \neq i}$
- $S_j \prec_{po} F \prec_{po} S_m : (W_j)^{x \neq i} \leq (W_m)^{x \neq i}$

D. Illustrative example

Fig. 2 illustrates how Axioms 7, 8, and 9 concur to properly specify the intended behavior for a classic example. In an architecture allowing nMCA stores, cumulative fences must be used to disallow the outcome r1 = NEW, r2 = 0, r3 = NEW, and r4=0.

In Fig. 2a, processors P_1 and P_2 performs stores S_1 and S_2, respectively. Their values might be observed by loads L_1 and L_2 in P_3, and by loads L_3 and L_4 in P_4. Let us provisionally assume that fences F_1 and F_2 only exhibit non-cumulative behavior, as captured by Axiom 8. Suppose an execution witness where L_1 observes S_1 in P_3 (i.e. $Val[L_1] = Val[S_1]$), resulting in r1=NEW, and where L_3 observes S_2 in P_4 (i.e. $Val[L_3] = Val[S_2]$), resulting in r3=NEW. If it turns out that L_2 has not observed S_2 in P_3 (i.e. $Val[L_2] = Val_b^0$), resulting in r2=0, and L_4 has not observed S_1 in P_4 (i.e. $Val[L_4] = Val_a^0$), resulting in r4=0, the former would imply that P_3 sees S_1 before S_2, while the latter would imply that P_4 sees S_2 before S_1. Thus, in such case, no order would exist for the stores. Such unacceptable behavior never happens to rMCA stores (for which a global linear order is guaranteed), but nMCA stores cannot guarantee proper behavior alone. That is why cumulative fences come into play.

Now let us show that, when F_1 and F_2 satisfy both Axioms 8 *and* 9, the execution witness r1=NEW, r2=0, r3=NEW, r4=0 is disallowed. Fig. 2b illustrates all the memory completion events corresponding to the stores and loads in Fig. 2a (commit events were omitted for simplicity). For each store, local completion is distinguished from remote completions by marking them as $\ominus$ and $\otimes$, respectively. Let us explain step-by-step how the axioms insert edges between events.

Suppose that P_3 sees S_1, but not S_2, i.e. $(W_1)_a^3 \leq (R_1)_a^3$ (1) and $(R_2)_b^3 \leq (W_2)_b^3$ (2). From Axiom 7, we have $\sigma_G(L_1) \neq \emptyset$ and $\sigma_G(L_2) = \emptyset$, leading to $Val[L_1] = Val[S_1]$ and $Val[L_2] = Val_b^0$, respectively, resulting in r1=NEW and r2=0. Since Axiom 8 ensures that $(R_1)_a^3 \leq (R_2)_b^3$ (3), the stores become ordered by transitivity, i.e. $(W_1)_a^3 \leq (W_2)_b^3$, meaning that it is as if P_3 sees S_1 before S_2.

Given that $\sigma_G(L_1) \neq \emptyset$ and $\sigma_G(L_2) = \emptyset$, Axiom 9 requires that $(W_1)_a^{i \neq 3} \leq (W_2)_b^{i \neq 3}$ (4). Suppose that P_4 sees S_2, i.e. $(W_2)_b^4 \leq (R_3)_b^4$ (5). From Axiom 7, we have $\sigma_G(L_3) \neq \emptyset$, leading to $Val[L_3] = Val[S_2]$, i.e. r3=NEW. Since Axiom 8 ensures that $(R_3)_b^4 \leq (R_4)_a^4$ (6) and $(W_1)_a^4 \leq (W_2)_b^4$ was enforced by F_1, we have $(W_1)_a^4 \leq (R_4)_a^4$ by transitivity. From Axiom 7, we have $\sigma_G(L_4) \neq \emptyset$, leading to $Val[L_4] = Val[S_1]$, i.e. r4=NEW, meaning that P_4 must have seen S_1 before S_2, as P_3 did! (Note that, in this case, since $\sigma_G(L_3) \neq \emptyset$ and $\sigma_G(L_4) \neq \emptyset$, Axiom 9 does not required F_2 to enforce orderings in other cores). Therefore, the unacceptable behavior was disallowed by the fences despite the nMCA stores.

IV. Observability Template

This section proposes a template that can accommodate most micro-architectures, because physical events are monitored only at the interface with buffers common to most dynamically scheduled pipelines and with buffers at the *first* cache level only[9]. It defines *which* physical events should be used as *proxies* for abstract events, and *where* they should be observed for proper axiom verification.

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

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

Fig. 3 shows the proposed template for a design with p processors, each accessing a private cache. Only the relevant flow of data is explicitly shown (the flow of addresses is omitted for simplicity). We assume the following types of buffers: (1) a *reorder buffer* (ROB) [17] (or similar structure for instruction commit in program order), (2) *store buffers* (or similar structures for keeping addresses and values of outstanding stores), (3) Incoming and outgoing *cache buffers* (for keeping local memory requests and replies), (4) Incoming and outgoing *inter-cache buffers* (for keeping coherence messages to or from other cores, either invalidate (update) requests or data replies). The template abstracts lower hierarchical levels, interconnection network, and coherence engines.

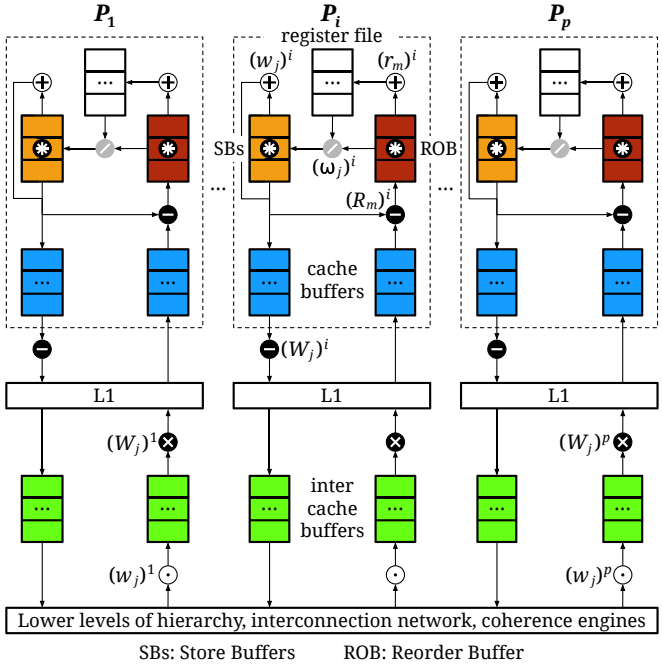


Fig. 3. Observability for a generic multicore chip design

Every *monitor* captures a physical event as a triple (op, a, v), where op is either write or read, a is a location, and v is a value (if any). Thus, the distinction between commit and completion events and between local and remote events can only be made by inserting monitors at adequate points. That is why we use distinct monitor types in Fig. 3, denoted as $\oplus$, $\ominus$, $\odot$, $\otimes$, $\oslash$, and $\circledast$. They indicate the relevant points for observing memory events in each processor domain. Let us show how a physical event observed by a monitor can be used as a proxy for a given abstract event.

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

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

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

V. Building Runtime Checkers

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

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

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

To face the huge number of valid execution witnesses resulting from a largely relaxed memory model, while preserving verification

quality, let us assume that a few properties are known to hold before shared memory verification is launched (because they were validated during processor design). The rationale is that, by checking for errors that are easier to find in advance, the dynamic checker can focus on efficiently uncovering more subtle errors.

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

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

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

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

Properties 1 and 2 allow a checker to use commit events as anchors for efficiently checking completion orderings[10].

Let us also assume that the dynamic scheduler properly manages load and store buffers such that it preserves program order between a store and a conflicting load, i.e. the value of the store is always made available before a conflicting load could read it, formally[11]:

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

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

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

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

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

Besides, we impose a couple of usual constraints [6], [8], [9] on test generation and execution. To keep a single set of monitors per core domain, we pragmatically constrain test execution, as follows:

Constraint 1. Each processor runs a single thread.

Albeit it is not always possible to fully distinguish between operations by relying only on the events they induce, this distinction is required when tracking the order in which conflicting stores complete in different processors. To enable that, test generation must be enforced in such a way that the values assigned by distinct *conflicting* stores are unique, as formalized below[12]:

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

VI. Experimental Evaluation

We implemented checkers for Section III's axioms, relying on Section IV's monitors, and built according to Section V's assumptions. We compared them with the most recently reported runtime checker, which is based on multiple relaxed scoreboards (MSB) [9] (we relied on the reported pseudo-code to implement a version of MSB compatible with our experimental infrastructure). We used gem5's infrastructure [20] for simulation and design representation (models `O3` out-of-order for CPU, `Ruby` for the memory system, and `simple` for the interconnect network). We adopted gem5's MOESI directory protocol, and a hierarchy with 64KB (4-way) private caches at L1 and a shared 4MB (16-way) cache at L2, all operating with the same block size (64 bytes). To synthesize high-quality tests, we employed a generator that was allowed to vary the number of shared locations in a test within a specified range (4, 8, 16, 32, 64, 128), but it was constrained to produce suites where all tests have fixed size, albeit distinct sizes were studied (1Ki, 2Ki, and 4Ki memory operations).

TABLE I
Studied design errors.

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

To quantify *false diagnoses*, we relied on designs containing no errors. Then we inserted different artificial errors to challenge the checkers by changing the FSMs that implement the coherence protocol (either by modifying the next state or precluding some due output action). Each faulty design contained a single, distinct error. The errors studied in the experiments are described in Table I. For a given test size, we launched the generator 12 times by exploiting different seeds, leading to 12 distinct test suites. We determined the fraction of them for which each checker raised errors in a correct design (i.e. false diagnoses). To determine the effort spent in an attempt to find a given error in a faulty design, we measured the runtime until the error was found or until generation was stopped, and we took the average on the set of all test suites. We obtained the overhead of our checker with respect to MSB by calculating the percentage of extra effort.

We targeted ARMv8, and evaluated the checkers over nMCA and rMCA designs. To obtain nMCA designs, we modified gem5's native (rMCA-compliant) coherence protocol, which was customized for allowing a core executing a store to forward its value to another core before it has received all invalidations [14], [15].

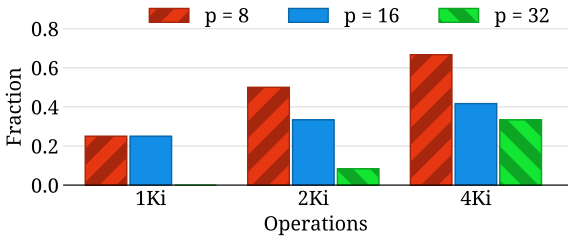


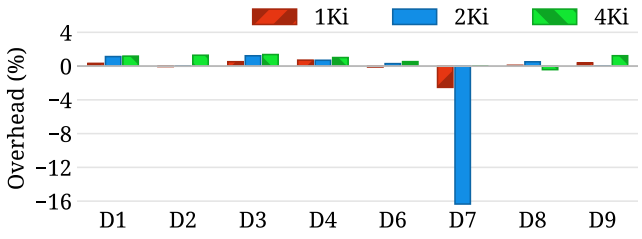
Fig. 4. Fraction of false MSB diagnoses for nMCA designs

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

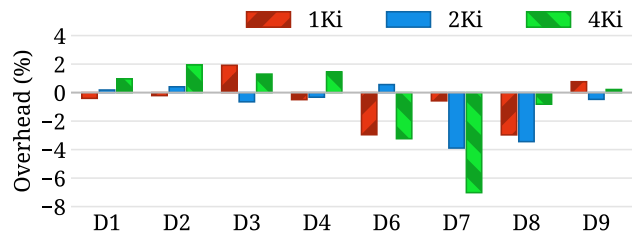
[11]There is no need for checking Axiom 6 under Property 3.

[12]Indeed, this is less restrictive than in related work (e.g. [6], [8]), where each store is assigned a unique value, which serves as its identifier.

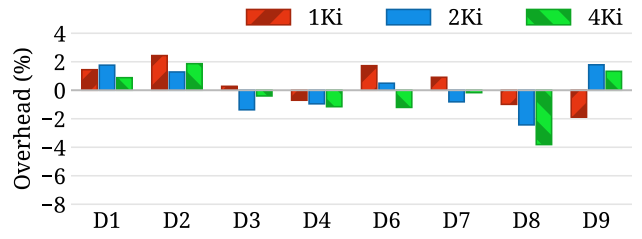
Fig. 4 shows the fraction of false MSB diagnoses for distinct test sizes and growing core counts, when handling correct designs. For a given core count, the fraction of false diagnoses significantly increases with test size, which is inconvenient, because larger test sizes are usually required to expose the most subtle errors in faulty designs. As a result of properly modeling memory behavior, our checker did not raise any false diagnosis *at all* under exactly the same conditions.



(a) 8-core designs



(b) 16-core designs



(c) 32-core designs

Fig. 5. Effort overhead for faulty rMCA designs

Then we used the proposed approach to build an rMCA-compliant checker, by adding extra constraints to Axiom 1 so as to restore (the originally relaxed) store atomicity. We compared our checker and MSB for detecting errors in faulty rMCA designs. Our checker was able to find all studied errors, but MSB was unable to find error D5. Fig. 5 shows the average overhead of our checker with respect to MSB for the designs where both checkers exposed errors. The maximum effort overhead observed was 2.5%, whereas the maximum effort reduction was 16%. Figure 5 indicates that the approach's versatility and improved verification quality may come at the expense of negligible additional effort, but often leads to effort reduction.

VII. Conclusions and Future Work

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

References

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

[2] S. Devadas, "Toward a coherent multicore memory model," *Computer*, no. 10, pp. 30–31, 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] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[5] W. W. Collier, *Reasoning About Parallel Architectures*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.

[6] C. Manovit and S. Hangal, "Completely verifying memory consistency of test program executions," in *IEEE Int. Symposium on High-Performance Computer Architecture (HPCA)*, 2006, pp. 166–175.

[7] O. Shacham, M. Wachs, A. Solomatnikov, A. Firoozshahian, S. Richardson, and M. Horowitz, "Verification of chip multiprocessor memory systems using a relaxed scoreboard," in *IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*, 2008, pp. 294–305.

[8] W. Hu, Y. Chen, T. Chen, C. Qian, and L. Li, "Linear Time Memory Consistency Verification," *IEEE Transactions on Computers*, vol. 61, no. 4, pp. 502–516, Apr 2012.

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

[10] 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, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 258–272.

[11] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU Concurrency: Weak Behaviours and Programming Assumptions," in *ACM Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 577–591.

[12] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux, "Automated synthesis of comprehensive memory model litmus test suites," in *ACM Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 661–675.

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

[14] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA," in *ACM Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 119–133.

[15] K. Gharachorloo, "Memory consistency models for shared-memory multiprocessors," Ph.D. dissertation, Stanford University, 1995.

[16] S. Adve and M. D. Hill, "Sufficient Conditions for Implementing the Data-Race-Free-1 Memory Model," University of Wisconsin - Madison, Tech. Rep., 1992.

[17] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann Publishers Inc., 2019.

[18] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, pp. 7:1–7:74, 2014.

[19] ARM. (2011) Cortex-A9 MPCore, Programmer Advice Notice, Read-after-Read Hazards. ARM Reference 761319. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.uan0004a/UAN0004A_a9_read_read.pdf

[20] N. Binkert et al., "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug 2011.