ILA-MCM: Integrating Memory Consistency Models with Instruction-Level Abstractions for Heterogeneous System-on-Chip Verification

Hongce Zhang, Caroline Trippel, Yatin A. Manerkar, Aarti Gupta, Margaret Martonosi, Sharad Malik
Princeton University, USA

***Abstract*—Modern Systems-on-Chip (SoCs) integrate heterogeneous compute elements ranging from non-programmable specialized accelerators to programmable CPUs and GPUs. To ensure correct system behavior, SoC verification techniques must account for inter-component interactions through shared memory, which necessitates reasoning about *memory consistency models* (MCMs) This paper presents ILA-MCM, a symbolic reasoning framework for automated SoC verification, where MCMs are integrated with *Instruction-Level Abstractions (ILAs)* that have been recently proposed to model architecture-level program-visible states and state updates in heterogeneous SoC components.**

ILA-MCM enables reasoning about system-wide properties that depend on functional state updates as well as ordering relations between them. Central to our approach is a novel *facet* abstraction, where a single program-visible variable is associated with potentially multiple facets that act as auxiliary state variables. Facets are updated by ILA "instructions," and the required orderings between these updates are captured by MCM axioms. Thus, facets provide a symbolic constraint-based integration between operational ILA models and axiomatic MCM specifications. We have implemented a prototype ILA-MCM framework and use it to demonstrate two verification applications in this paper: (a) finding a known bug in an accelerator-based SoC, plus a new potential bug under a weaker MCM, and (b) checking that a recently proposed low-level GPU hardware implementation is correct with respect to a high-level ILA-MCM specification.

I. Introduction

Systems-on-Chip (SoCs) integrate specialized hardware to meet the power-performance requirements posed by emerging applications. Specialized hardware can be programmable (e.g., Graphics Processing Units or GPUs) or non-programmable (e.g., an AES cryptographic accelerator). They outperform general purpose processors in specific domains like machine learning [1], scientific computation [2], and cryptographic operations [3]. The multiple processing units in an SoC typically run concurrently. This concurrency can be difficult to reason about, leading to design and implementation bugs in functional correctness as well as security. Furthermore, when SoC components interact via shared memory or memory-mapped input and output (MMIO), one also needs to reason about *memory consistency models* (MCMs). Although programmers generally find it easier to think about concurrent code with sequentially consistent (SC) ordering semantics, modern instruction set architectures (ISAs) have weaker MCMs in an effort to achieve better performance and scalability.

This work was supported in part by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

This work was supported in part by the National Science Foundation, XPS Program, Grant No. 1628926.

Previous MCM verification efforts have focused on modeling and analyzing MCMs at different levels of the software/hardware stack in parallel systems [4–11]. These approaches typically use small parallel programs, called *litmus tests*, for reasoning about the MCMs themselves. They focus on *ordering relations* between simple instructions, rather than on symbolic reasoning of complex control and data flow in programs, which is often needed in SoC verification. Moreover, none of these efforts consider non-programmable hardware accelerators, which may not have an ISA.

Recently, an instruction-centric operational model for heterogeneous SoC components has been proposed, called an Instruction-Level Abstraction (ILA) [12]. Analogous to a processor ISA, an ILA models a hardware component's program-visible states and their updates in the form of instructions. This provides a well-defined interface between sequential software and the underlying hardware component. For an accelerator, its ILA instructions correspond to commands at its interface. ILAs have been successfully generated (using semi-automated synthesis-based techniques) for many accelerators in practice [12–14]. In the rest of this paper, we use "instructions" to denote ILA instructions, which correspond to instructions in a processor ISA or to derived instructions for an accelerator.

An ILA can uniformly model *rich* instruction semantics (i.e., including control and data flow) of a *single* processing unit, e.g., a processor or an accelerator. Although existing MCM specifications and verifiers are well-suited for representing orderings between memory operations of multiple processing units, they lack such rich instruction models. We show that for general SoC verification, it is essential to reason about *both* rich instructions in heterogeneous components and memory orderings between them.

In this paper, we address this central challenge by proposing a general symbolic framework called ILA-MCM, shown in Figure 1. In this framework, each processing unit in an SoC, such as a programmable processor or an accelerator, is uniformly represented by an ILA. The MCM is described using axioms, as in previous efforts [4–11], but is *integrated with the ILA operational models*. This enables our ILA-MCM framework to reason about functional state updates in instructions as well as the effects of MCMs, thereby supporting expressive properties involving both states and orderings for SoC verification.

A novel feature of our ILA-MCM framework is the *facet* abstraction, where a single program variable in an instruction can be associated with multiple auxiliary state variables called facets in the verification model. Facets are useful for modeling

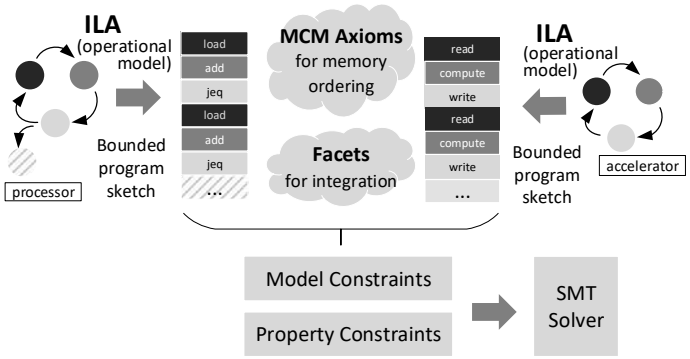

Fig. 1. The ILA-MCM Framework for heterogeneous SoC verification

memory subsystems and consistency effects, where different observers in an SoC may see logically distinct values of the same program-visible variable. The allowed values of facets are constrained by the operational semantics of the instructions as well as the memory consistency axioms. Thus, facets form a critical link between operational ILA models and axiomatic MCM specifications.

Another feature is that our verification procedure supports both operational and axiomatic models in general. (For example, our second application uses a low-level operational model for memory consistency.) The executions of operational models (e.g., ILAs) are based on a *program sketch* [15][1], which depends on the property to be verified. This creates *symbolic trace events* (events, in short). Each event is guarded by a condition and updates the state in an ILA or a facet. The axioms are then instantiated, which may create additional events or impose *happens-before* [16] ordering relations between events. We refer to these sets of constraints as the *model constraints*. Finally, we add *property constraints* that refer to states and ordering requirements for verification.

We use standard theories in first order logic to capture all constraints, including the semantics of instructions in a program and happens-before ordering relations between events. The formula comprising all constraints is checked by a Satisfiability Modulo Theory (SMT) solver [17]. Our framework supports diverse verification tasks formulated as SMT queries, including finding bugs (via falsification) or proving correctness (via verification condition generation). We have implemented a prototype ILA-MCM framework and demonstrate its use in two challenging SoC verification applications in this paper.

To summarize, this paper makes the following contributions:

- **ILA-MCM framework:** We propose a framework that combines operational models for processing cores (including accelerators) with axiomatic memory consistency models to enable SMT-based reasoning of complex interactions between hardware, software, and memory subsystems in heterogeneous SoCs.
- ***Facet* abstraction:** We propose the facet abstraction, where a single program-visible state variable can be associated with multiple logically-distinct variables, to represent updates on program-visible states with memory consistency effects. The facets provide the basis for a constraint-based integration of ILAs with MCMs.
- **Evaluation on real-world SoCs designs:** First, we show an application of the ILA-MCM framework for finding security bugs in SoC firmware [18], where our support for expressive properties enables finding a malicious exploit from a program sketch. Second, we show an application for checking correctness of a low-level GPU hardware implementation [19] against a high-level ILA-MCM specification, where our instruction-centric approach enables its decomposition into simpler verification tasks.

An overview of various components in the ILA-MCM framework is shown in Figure 2, annotated by the section numbers that describe these components. We start by introducing the relevant background on ILAs and MCMs.

II. Background

A. Instruction-Level Abstraction (ILA)

An ILA is a uniform abstraction for hardware accelerators as well as general-purpose/specialized programmable processors [12]. It is an operational model that captures updates by hardware to *program-visible states* (i.e., the states that are accessible or observable via a user-facing program instruction). It can be viewed as a generalization of the processor ISA in the heterogeneous context, where the instructions for accelerators are defined as the commands on their interface that update program-visible states. In an ILA, each instruction has a decode condition, and the instruction executes only when this condition is true. An ILA also supports hierarchy, where an instruction at a high level can be represented as a *sequence* of *child instructions* at a lower level, as shown in Figure 2 for `Instr A` of `ILA1` (under the "ILAs" column). Thus, the granularity of ILA instructions can vary, ranging from processor instructions to software functions. Furthermore, an ILA is used for modeling a sequential thread of control, while parallelism is modeled using multiple such threads.

B. Memory Consistency Model (MCM)

An MCM provides a specification to a programmer of the order in which memory operations appear to execute [20]. Sequential consistency (SC), defined by Lamport [21], specifies that: (1) memory accesses preserve the order within each thread of a program, and (2) across threads, there is an order of accesses that every observer agrees upon. Despite the intuition of SC, nearly all modern ISAs adopt MCMs weaker than SC. A weak MCM allows certain memory accesses to be reordered within a program, and supplies fences or other synchronization mechanisms to enforce required orders when necessary. For example, the Total Store Order (TSO) model allows a load to be reordered with earlier stores that access a different address to allow the store-buffer optimization [22].

Figure 3 illustrates the effects of MCMs on a small multi-threaded program with a proposed outcome, called a litmus test. In this litmus test, each thread executes a store (`st`)

[1] Similar to automated program synthesis, the "holes" in our program sketch are filled in by a solver.

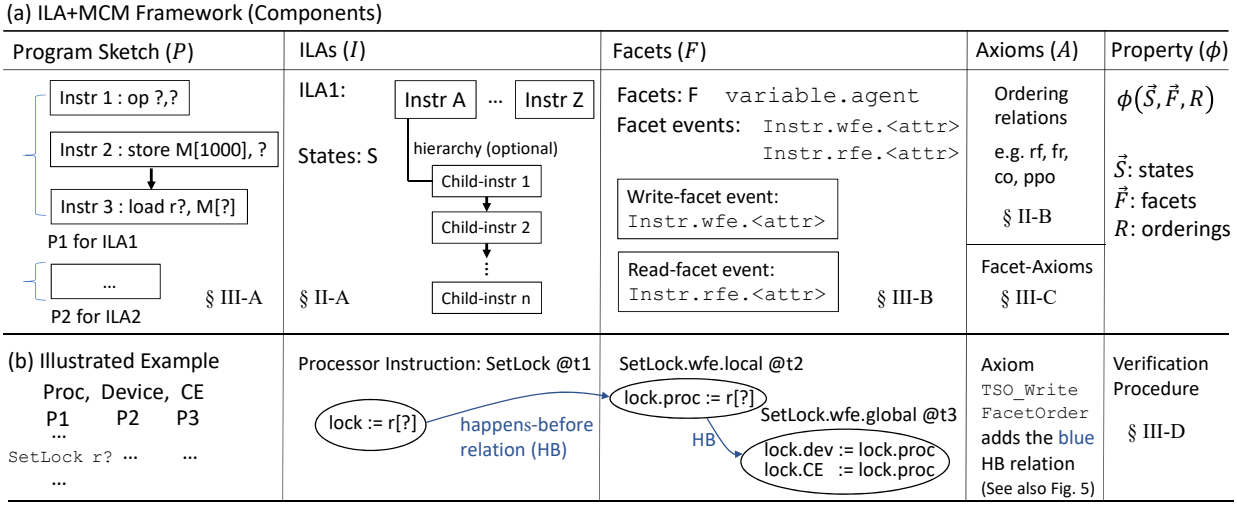

Fig. 2. Components of the ILA-MCM framework, with example fragments.

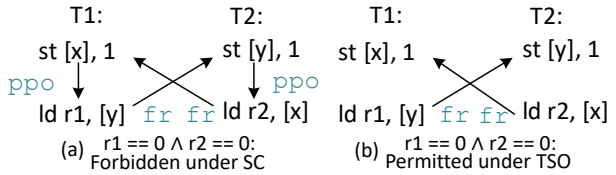

Fig. 3. A forbidden outcome in SC can become permitted under a weaker MCM. Arrows show the ordering relations (in blue) between instructions.

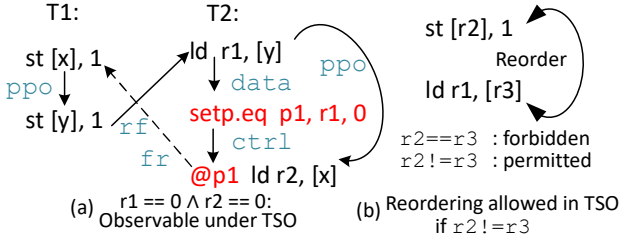

Fig. 4. Examples of conditional orderings. (a) `@p1 ld` executes iff `p1` is `true`, i.e., iff `r1==0` (setting/using predicate `p1` is marked in red). (b) In TSO, store-to-load reordering is allowed if the addresses are different.

and then a load (`ld`) instruction, where all memory locations and registers are initially 0. Figure 3(a) assumes the SC MCM, and thus forbids a program outcome where both load instructions return 0. This is evident in a cycle of edges that comprise the preserved program order between the store and load instructions (shown as *ppo* edges) and the order between the read in one thread and the write in the other (shown by from-read (*fr*) edges). In contrast, under TSO (Figure 3(b)), the *ppo* edges are removed (since a read can be reordered with an earlier write), so the proposed outcome is permitted since there is no cycle. In general, MCMs also consider the *co* edge (coherence order between writes to the same address) and the *rf* edge (reads-from order from a write to a load which reads from that value).

C. Gaps in Prior Work

Despite a rich history of prior work in MCM verification, they lack some key capabilities described below.

Symbolic Reasoning with Conditional Orderings. Our main goal is to support general verification of SoC software and hardware. However, most prior efforts in MCM verification rely upon an explicit enumeration over addresses, data, and conditional predicates that may affect orderings between memory operations. Specifically, we consider the following two types of conditional orderings: ❶ relations involving predicated instructions or instructions after branches, and ❷ relations involving address/data-dependent values.

For example, Figure 4(a) shows ❶, with a predicate `p1` on the last load instruction in thread `T2`. Note that the existence of the load event and the related *fr* edge (shown as a dashed arrow) are *control-dependent*. If this control-dependency is ignored, the analysis will *incorrectly* deduce that the graph is cyclic, i.e., the outcome is unobservable. Figure 4(b) shows an example for case ❷, where reordering is allowed only when the addresses in registers `r2` and `r3` are different.

In prior MCM efforts based on relational models, e.g., using Alloy [5] or Check tools [7–11], the addresses and data are modeled by relational predicates, e.g., whether two addresses are the same. However, such relations have to be pre-specified and are not explored symbolically in the solver. Similarly, Herd uses enumeration over all possible values of relevant addresses/data. In contrast, ILA-MCM uses symbolic reasoning to represent ordering relations dependent on complex contro/data flow and avoids explicit enumeration.

Rich Instruction-Centric Models. Most previous efforts in MCM verification focus on ordering relations between instructions, rather than on updates of program-visible states. For example, arithmetic instructions are abstracted away in relational models [5]. In Herd [4], the instructions are hard-coded and do not model bit-precise hardware (e.g., there is no register overflow behavior). Our goal is to support SoC verification by modeling rich instruction semantics for processors as well as non-programmable accelerators, which is required for reasoning about general (not just litmus) programs.

Expressive Properties. MCM verification has typically focused on specifying orderings and litmus tests, while program/processor verification has focused on state-based veri-

fication or control-oriented properties. We aim to support SoC verification using a wide range of expressive properties that can refer to *both* states and orderings.

III. ILA-MCM Framework

We now provide details of the main components of our ILA-MCM framework (shown in Figure 2): program sketches, facets, axioms, and verification procedure.

A. Program Sketch

We leverage existing work on programming by sketches [23, 24] to synthesize a program that would exercise a bug or abstractly capture unbounded executions. Our program sketch comprises: (1) a set of partially-specified state updates in instructions (and any child instructions), and (2) a partial order on them. Holes (shown as question marks in Figure 2) are allowed in the sketch. These are filled in by the SMT solver during verification. Examples of holes include symbolic values (e.g., content of a memory location) or fields in an instruction encoding (e.g., address/data field of the `store` and `load` in Figure 2).

The program sketch, which needs to be provided by the user, typically depends on the correctness property. Although a program sketch has a bounded number of instructions, one can use an outer procedure to iteratively increase the bound, to perform a deeper search for bugs or for a proof by induction using given invariants. In the first column in Figure 2(b), the example considers an SoC with a processor, a device, and a cryptographic engine (CE). Thus, there are three program sketches (P1, P2, P3) and a `SetLock` instruction is illustrated in the program sketch (P1) for the processor. The second column (under ILA) shows the related event, which updates the lock variable by the value of some register (left as a hole `r?`) and associates a symbolic timestamp `t1` with the event.

B. The Facet Abstraction

To reason about the interactions between SoC components via shared memory, we need to establish a relation between program variables in instructions of different ILA models via axioms in MCMs. We model this using a novel abstraction described below.

1) State Variables for Facets: Facets are auxiliary variables associated with a *shared* program-visible state variable that can be observed by an "agent," which may be a thread, a physical structure or a processing core/accelerator, depending on the ILA modeling granularity. Facets reflect the fact that different agents may observe distinct values of the same shared variable in different orders. For example, the store-to-load reordering in TSO can result in the load seeing the new value from the store earlier than instructions on another thread. In general, each agent can potentially have its own facet for a shared variable. In our experience, this per-agent-facet is general enough to model weak consistency behaviors. (More facets can be added if one wishes to model memory consistency at the microarchitecture level, e.g., with store-buffers or caches, etc.)

We use the notation *variable.agent* for the facet that corresponds to a specific agent's view of a given program variable. For the example considered in Figure 2(b), suppose there is an on-chip interconnect between the three components, and that there is a register in the device denoting a lock. The device observes its value by directly reading the register, which is regarded as the facet of the device (denoted *lock.dev*). The device provides a memory-mapped interface, where other agents can access the lock register as if accessing a memory location. We model the lock register seen by the other agents as facets, denoted *lock.proc* and *lock.CE*, respectively.

2) State Updates for Facets: Continuing with our example, the ILA instruction `SetLock` on the processor can update the lock by writing to the memory-mapped address of the lock register in the device. The new value may first appear in the processor's local buffer, then go into a cache, and through the interconnect, propagate to the device and finally update the device's register. This could result in different agents seeing different values in different orders. We model this by creating new events: write-facet events to update facets, and read-facet events to read facets.

For example, TSO can be modeled such that each agent has a facet for a shared program variable. A store instruction creates two write-facet events, one to its own facet (local write-facet event) and the other to all other facets (global write-facet event). A load instruction corresponds to one read-facet event, since it only needs to read from its own facet. In general, any instructions or child-instructions accessing shared variables can have associated facet events. The values that facet read/write events use for updates are derived from the ILA instruction semantics, while the orderings of facet read/write events are specified by the facet-axioms in the MCM. We use the notation *instr.wfe/rfe.<attr>* to refer to the write-facet events (wfe) or read-facet events (rfe), related to a given instruction (*instr*), with a given attribute *<attr>*. In the TSO model, *<attr>* can be $local$ or $global$. The example in Figure 2(b) shows two write-facet events (under Facets) related to the `SetLock` instruction under the TSO model.

C. Facet-Axioms for Integrating ILAs and MCMs

So far, we have described facets as state variables, and new facet events associated with ILA instructions that update or read them. The orderings between these events are specified by MCM axioms. For SC and TSO, the complete set of facet-axioms can be found in the Appendix. We highlight some fragments of these in Figure 5. Note that we uniformly use happens-before relations (denoted as `HB`) to specify orderings between events. In the SC model (top part), all facet read/write events are synchronous (i.e., these events occur at the same time) with the instructions (lines 1-2). In the TSO model (lower part), the two write-facet events (local or global) of a store instruction happen after the instruction and follow the program order (lines 3-9). These axioms are similar to those used in prior work, e.g., in the μspec TSO model [7], except that facet-axioms relate instructions with facet read/write events, while μspec axioms relate instructions

```
 1: Axiom SC_WriteFacetOrder
 2: forall w:WRITE | Sync[ w , w.wfe.global ]
    …
 3: Axiom TSO_WriteFacetOrder
 4: forall w:WRITE | HB[ w , w.wfe.local ]  /\
 5:   HB[ w.wfe.local , w.wfe.global ]
 6: Axiom TSO_Store
 7: forall w1:WRITE | forall w2: WRITE (not w1) |
 8:   PO[ w1, w2 ] => HB[ w1.wfe.local, w2.wfe.local]
 9:      /\ HB[ w1.wfe.global, w2.wfe.global]
    …
10: Axiom RF_CO_FR
11: forall r:READ | exists w:WRITE |
12:   SameAddress[w,r] /\ SameData[w,r]/\ w.decode /\RF[w,r]/\(
13:      forall w2:WRITE (not w) | ( SameAddress[w,w2] /\
14:         w2.decode )=> CO[w2,w] \/ FR[r,w2] )
15: Define RF[ w, r] := …
16: Define CO[w1,w2] := …
17: Define FR[ r, w] := …
```

Fig. 5. SC and TSO axioms (fragments)

with microarchitectural structures like pipeline stages and caches. Further, axioms for other MCMs can be similarly defined. We have designed these axioms by hand (similar to prior MCM work); addressing their correctness is beyond the scope of this work.

The main highlight of the facet-axioms is that the relations over facet events in the MCM are linked with control/data flow in the ILA instructions via predicates interpreted over ILA state variables and facets. Consider the `RF_CO_FR` axiom (lines 10-14), which states that: (a) all read events should read from some executed write event with the same address, and the data values of read and write should match, (b) if a read r reads from a write w, any other executed write w_2 should not interfere. Here, the predicates `SameAddress` and `SameData` are interpreted over ILA state variables and facets. Similarly, the symbolic decode condition of an instruction (denoted *instruction.decode*) is a predicate over ILA state variables. Note also that the definitions of *rf*, *fr*, and *co* edges are based on the happens-before relation over facet-events.

D. ILA-MCM Verification Procedure

Our verification procedure is shown in Algorithm 1. Among its inputs, the first is a program sketch $P(T, R)$, where T is a set of instances[2] of partially-specified (child-) instructions, and R is a partial order. Other inputs are a set of ILAs I, the axioms A, and a property ϕ. For each possible instruction instance, the algorithm creates a trace step (simply called step) using the instruction semantics[3] (line 5). We also associate a symbolic timestamp with the step, encoded as an integer (t_a for step a). Values of timestamps only reflect relative orderings. Recall that the instructions/child-instructions may lead to facet read/write events, and steps are also created for these events (lines 6-8). Next, any happens-before orderings in the program sketch are interpreted as a less-than relation on the associated timestamps (line 10). Then, we instantiate the quantifiers and interpret the predicates in the axioms over the set of steps (line 11), and add the negation of the property (line 12). Finally, the set of constraints is checked by an SMT solver. (Our prototype uses Z3 [27].) If the constraints are satisfiable, we get a counterexample in the form of an event trace; otherwise, the property is valid within the space allowed by the program sketch. To verify unbounded correctness, we can check whether given invariants are inductive and use abstractions to model nondeterministic environments, as discussed later in Section IV-B.

[2]Multiple occurrences of the same (child-) instruction are regarded as separate instances in a trace.

[3]Although not shown here, we use a concurrent static single assignment (CSSA) encoding [25, 26], where uses of shared state variables are encoded as π-variables and updates to them are encoded as new definitions.

Algorithm 1 ILA-MCM Verification Procedure

```
 1: procedure VERIFY(P(T,R), I, A, φ)
 2:     ▷ P(T,R): program sketch P, where T is a set of instances
        of (child-) instructions and R is a partial order, I: set of
        ILAs, A: axioms, φ: property
 3:     C ← ⊤                                   ▷ C is set of constraints
 4:     for each ts ∈ T do
 5:         C ← C ∧ CreateStep(ts, I)
 6:         T' ← AssocFacetEvent(ts, A)          ▷ Get facet-events
 7:         for each ts' ∈ T' do
 8:             C ← C ∧ CreateStep(ts', I)
 9:     for each a → b ∈ R do
10:         C ← C ∧ t_a < t_b                    ▷ Orders are on timestamps
11:     C ← C ∧ InstantiateAxioms(A)
12:     C ← C ∧ ¬ φ
13:     if SMTCheck(C) = SAT then
14:         return INVALID, GetModel(C)
15:     else return VALID
```

IV. Verification Applications

A. Security Bug in a Firmware Load Protocol

1) System Overview: The SoC [18] used in this application consists of a processor, a device, and a cryptographic accelerator engine (CE). The processor runs a driver that loads a firmware image onto the device. The CE is responsible for authenticating the image before it can be used by the device. The SoC has a system memory (SM) that all three agents can access, and an isolated memory (IM) that can only be written by the device but is readable by both the device and the CE. The threat model assumes that the driver on the processor can be compromised. The attacker's goal is to fool the device into running a malicious firmware image that does not carry a correct signature.

2) ILAs and Instructions: The first step is to construct an ILA for each of the agents: the processor, the device, and the CE. The set of instructions and child instructions are shown in Figure 6(a) (along with a legend). The processor uses store operations to send commands to the memory-mapped device or the accelerator interface, and can query the status via reading through this interface. The ILA instructions in the processor (device driver) are `Send_Command_Reset`, `Store_Firmware`, or `Send_Command_Load`. The processor also has a `Receive_Report` instruction that, when triggered by an interrupt, reads from the device's status register to learn the result of firmware image authentication. The device ILA has three instructions: `Reset`, `Load` and `Handle_CE_Response`. The CE ILA has only one instruction (`Authentication`), which handles the authentication request.

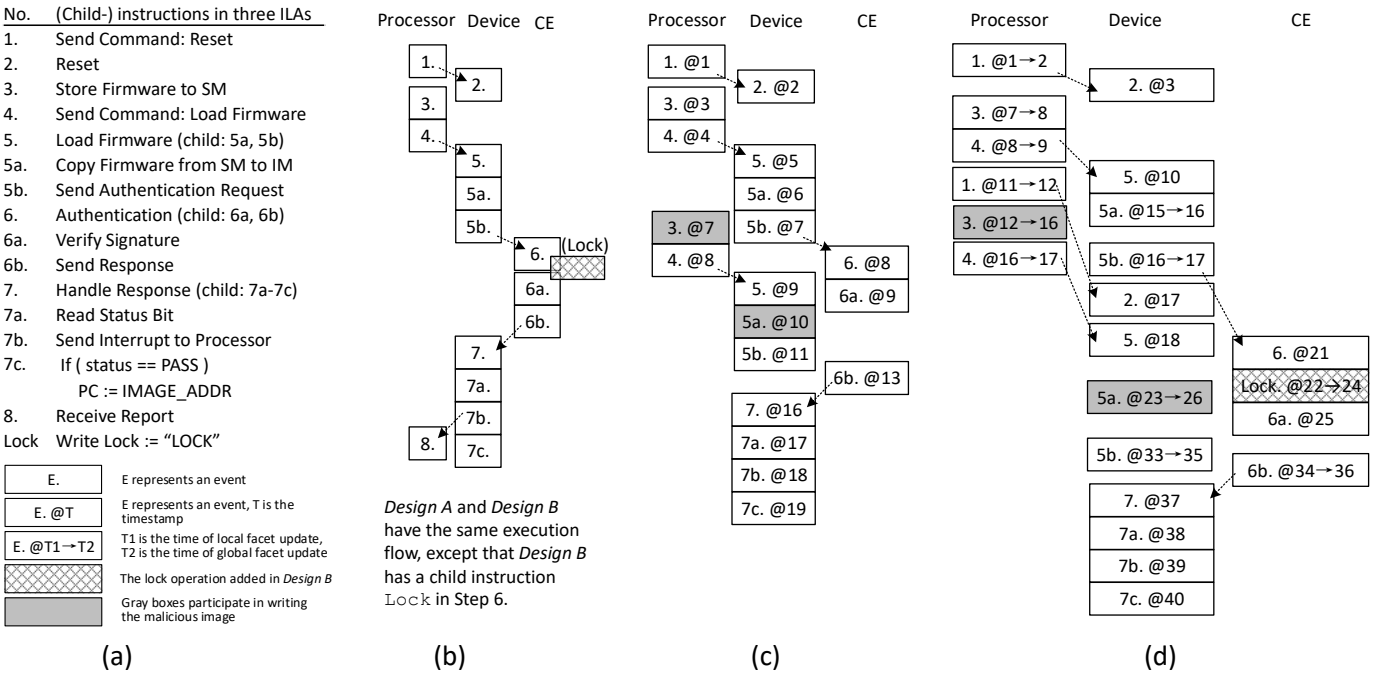

Fig. 6. (a) Instructions/child-instructions in the ILAs, plus legend. (b) Intended execution flow for *Designs A* and *B*, where a dashed arrow indicates an agent triggering an instruction in another agent via instruction-decode conditions. (c) Malicious exploit for *Design A* under SC, with event timestamps (`@T`) generated by the SMT solver. (d) Malicious exploit for *Design B* under TSO, with timestamps for local/global facet updates also generated by the SMT solver.

The intended execution flow of these instructions is shown in Figure 6(b). First, the driver sends a `Reset` command to the device by writing into the command register and the device performs reset (Step 1 and 2). The driver stores the firmware image in a dedicated region in SM (3) and invokes the device (4). Upon receiving the `Load_Firmware` command (5), the device copies the firmware image into its IM (child-instruction 5a) and sends an authentication request to the CE (5b). The CE checks the signature of the image in IM (6a), stores the result into its register and signals the device of its completion (6b). The device will read the verification result from the CE's address space (7a) and report the result to the driver (7b). If the result indicates that the image is authenticated, the device sets its own program counter to point to the firmware location in IM and starts its execution from there (7c). Finally, the processor handles the interrupt and knows that the firmware image has been loaded (8).

We refer to the above implementation as *Design A*, which is known to have a time-of-check to time-of-use (TOCTOU) vulnerability. Prior work originally identified and presented a solution to this vulnerability, namely *Design B* [18], where the device protects IM contents with a lock that is accessible only by the device and the CE. Once locked, the image stored in IM cannot be changed. Our ILA model for *Design B* is similar to *Design A*, except that the CE has an extra child-instruction `Lock` in ILA instruction 6 which enables the lock.

3) Program Sketch: We created a program sketch based on the instructions shown in Figure 6(a), where the solver explores which instructions to include in the malicious exploit by creating a hole for the decode condition of each instruction. Further, the values and addresses of the stores by the driver are left as holes in the program sketch.

4) Facets and Axioms: In this application, we consider two possible MCMs: SC and TSO. We use facets and axioms (shown in the Appendix) to model the MCMs.

5) The Property: The SoC should ensure the following safety property ϕ: $(DevPC = FwAddr) \rightarrow Check(IM[FwAddr]) \neq FAIL$. It says that when the device's program counter points to the region holding the firmware image, the image should not be malicious. Our verification procedure aims to synthesize an exploit that violates this property.

6) Results: Under the SC model, our verification procedure successfully reproduced the known malicious exploit [18] for *Design A* in 3.5 seconds, with a bound of 30 ILA instructions. The malicious exploit is shown in Figure 6(c), where the timestamps (`@T`) found by the SMT solver are shown for each event. Note that the correct image is authenticated, but the firmware overwrites it with a malicious image, which is then executed. This is a TOCTOU vulnerability.

Design B is intended to fix the above issue and works correctly under the SC model. However, under the TSO model, our verification procedure found a malicious exploit in 6.5 seconds, with a 32-instruction bound. To the best of our knowledge, this TSO-based vulnerability was not known before. The resulting trace is shown in Figure 6(d), where the essential problem is around timestamp 22 to 24. Although the CE updates the device's lock register at time 22, the device does not see this update until later. As shown, at time 23, the device overwrites the firmware with a malicious image. This bug can be fixed by adding a fence on the CE to ensure that the device sees the lock before the CE proceeds to authenticate.

B. Verifying Correctness of a GPU Implementation

Graphics Processing Units (GPUs) often use very weak consistency models that allow for a large amount of buffering and reordering of memory requests, to provide mitigation of high memory latency. An operational model of a GPU implementation is discussed by Wickerson et al. [19]. The implementation is intended to be compliant with OpenCL [28] (a variant of the heterogeneous-race-free (HRF) MCM [29]), with an extension called *remote scope promotion (RSP)* proposed by AMD. Under OpenCL, all programs must be free of data races (i.e., two unsynchronized accesses to the same address with at least one write); the behavior is undefined otherwise. Synchronization can be achieved by an acquire-load reading from a release-store with or promoted to a matched scope.

We aim to verify that the given hardware implementation is correct with respect to a high-level specification model that we build in ILA-MCM. We should mention that our specification is actually *more conservative* than the language-level OpenCL+RSP model described by Wickerson et al. – developing an equivalent ILA-MCM model for the latter is left to future work.

1) ILA-MCM Specification Model: This model comprises the functions of store, load, and atomic increment operations, plus the ordering relations they enforce. Each operation may have additional attributes that affect the ordering relations: (a) whether it is a release (for a store), an acquire (for a load), neither, or both, (b) the scope of the synchronization, and (c) whether it promotes the scope of a remote synchronization. We model these operations using ILA instructions, where different attributes lead to different instructions, e.g., store-relaxed and store-release are modeled as two distinct instructions. They have the same state updates, but the difference in their orderings is captured by the associated MCM axioms.

The system has a hierarchical structure comprising M devices, each device with N workgroups, with a workgroup having L threads. For a shared program variable, each thread possesses a facet, and additionally each workgroup (and each device) also has a facet. A store instruction will first update the facet of its own thread (TH-facet update), then the facet of its workgroup (WG-facet update) and the device facet (DV-facet update). A load instruction will have a TH-facet-read event (and potentially WG-facet-read and DV-facet-read events).

For each instruction, we use facet-axioms to model the enforced ordering requirements. For example, for the store-release (device scope, no remote promotion) instruction $\texttt{store}_{DV,N}$, one of its axioms is shown in Figure 7(a). It says that for a $\texttt{store}_{DV,N}$ instruction s_1, for all the other store instructions s_2 different from s_1, if they are on the same workgroup and there is a happens-before relation on their WG-facet updates, then their DV-facet update events also follow a happens-before relation. For each instruction, there can be multiple axioms specifying its ordering relations with different types of instructions under different conditions.

2) SoC Implementation: The implementation model, from Wickerson et al. [19], is fully operational (does not require facets or axioms). It contains a number of GPUs, where each

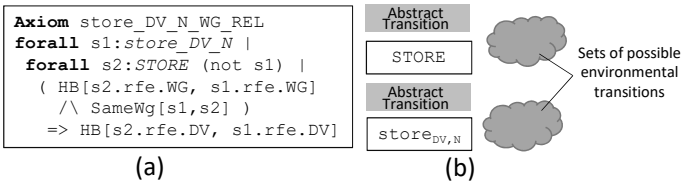

Fig. 7. (a) An axiom for instruction $\texttt{store}_{DV,N}$ (b) related program sketch

GPU performs a series of operations to achieve the effect of an instruction in the high-level specification. These operations are modeled as child instructions, which make use of the physical locks, FIFOs, and caches to guarantee correct data transfers and orderings.

We model 13 child instructions. Some examples are LD (load from L1 cache to register), ST (store from register to L1 cache), $\texttt{FLU}_{L1}\texttt{WG}$ (flush the L1 cache in its workgroup), $\texttt{INV}_{L1}\texttt{WG}$ (invalidate L1 cache of its workgroup). Inside a GPU, there are also other environmental transitions, e.g., a store may later trigger a cacheline flush. We model these state changes by child instructions as well.

3) Verification: We verify correctness of the implementation by checking that: (1) the program variables are updated to the same values as in the specification, and (2) the ordering of the updates is correct. The first check corresponds to functional equivalence checking between child-instructions on the GPU and the instructions in an ILA-MCM model, which can be handled using prior techniques [12]. Therefore, we focus here on the second check, where we use our facet-axioms as properties, and check if it is possible to synthesize a sequence of child instructions whose execution can violate the property. To ensure correctness using bounded traces, we need to further use invariants and abstractions.

We perform verification as follows. First we choose an instruction from the ILA-MCM specification model, collect axioms that refer to this instruction, and verify these axioms one by one. Since our facets and axioms are all instruction-centric, this instruction-based decomposition of the overall verification problem is directly enabled by our ILA-MCM framework, thereby providing a potential scalability benefit in comparison to handling all axioms monolithically.

An axiom may refer to other related instructions. For example, in the axiom in Figure 7(a) for the $\texttt{store}_{DV,N}$ instruction, there is a reference to another store instruction (of any type). We build a program sketch accordingly, as shown in Figure 7(b) for this example. Here, each of the two white boxes ($\texttt{store}_{DV,N}$ and STORE) denotes the sequence of child instructions that implement the high-level specification instruction, respectively. Since GPU operations may trigger environment transitions, we also add them in our program sketch. Finally, we add abstract transitions before and between the two sequences of child instructions. An abstract transition is allowed to update the state to any value (i.e., it is a $havoc$ operation), which is constrained subsequently by given invariants. The given invariants are checked separately on all child instructions (some require checking on all pairs). An example invariant is that the tail of a FIFO never passes the

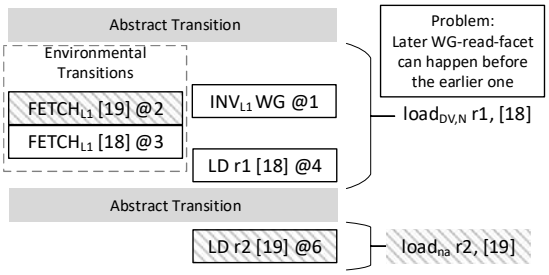

Fig. 8. The counterexample found for $\texttt{load}_{DV,N}$, where the addresses, timestamps and specific environmental transitions are generated by the solver

head, i.e., the FIFO does not underflow. In the future, we aim to maintain a library of invariants and abstract transitions for reuse. Further, the ILA-MCM verification procedure could be integrated with a general-purpose theorem prover to formally ensure their soundness and aid bookkeeping.

Although our ILA-MCM specifications are parametric, we do not perform parametric verification here, since the GPU implementation is fixed by a concrete system configuration (M, N, L). We currently performed verification for $M, N, L = 2$ and $M, N, L = 3$.

4) Results: For the original GPU implementation verified by Wickerson et al. [19], our verification failed with counterexamples for the following 5 instructions: $\texttt{load}_{DV,N}$, $\texttt{load}_{DV,R}$, $\texttt{store}_{DV,R}$, $\texttt{fetch\_inc}_{DV,N}$, and $\texttt{fetch\_inc}_{DV,R}$. Among them, $\texttt{load}_{DV,N}$, $\texttt{fetch\_inc}_{DV,N}$, $\texttt{store}_{DV,R}$ match with the buggy scenarios discussed in the previous work [19]. Specifically, Figure 8 shows a buggy trace that we found for instruction $\texttt{load}_{DV,N}$, where the facet-read event of the later non-atomic load instruction comes earlier than the facet-read event of the load-acquire instruction. This violates the load-acquire semantics. On the other hand, the counterexamples for $\texttt{fetch\_inc}_{DV,R}$ and $\texttt{load}_{DV,R}$ are false positives, since these traces cannot be extended to litmus tests with a property violation without having data races (prohibited by OpenCL). Interestingly, the proposed changes by Wickerson et al. to the compiler mappings of OpenCL+RSP operations strengthened the ordering guarantees of the hardware operations to match our ILA-MCM model. Under their new compiler mappings, we successfully validated that the hardware implementation is compliant with our ILA-MCM model. This validation was completed in 14 minutes 9 seconds (for $M, N, L = 3$) on a laptop with a 2.8GHz Core-i5 processor and 16GB memory.

V. Related Work

A. Hardware Specification and Verification

A number of formal hardware abstractions have been developed that enable verification. Kami [30, 31] is a Coq-based framework that supports hardware design and verification in Bluespec. In comparison to Kami, ILA-MCM is an ISA-level abstraction that provides the interface between hardware and software. In addition to verifying hardware, it can also be used for verifying correctness/security of software interacting with accelerators, as demonstrated in our paper. Furthermore, it can reason about a wide range of memory consistency behaviors, including SC, TSO, and HRF. In contrast, currently Kami has only been applied for SC. Finally, the ILA-MCM framework targets automated reasoning using SMT solvers, in contrast to interactive theorem-proving in Kami.

ISA-Formal [32, 33] has been developed to formally model and verify ARM processors. As its name suggests, it is an ISA-level model. However, it has not been applied to accelerators or other heterogeneous SoC components. Further, as far as we know, it has not been integrated with MCMs to reason about multicore memory consistency.

B. MCM and Program Verification

We have already discussed prior MCM verification tools and techniques. For reasoning about general concurrent programs, there are many related efforts in weak consistency models [34–36], logics [37, 38], and verification tools [39–41]. Here we discuss details of specific related ideas.

1) Facets vs. ViCLs: In the Check tools [7–11], the Value in Cache Lifetime (ViCL) abstraction has been proposed to capture cache occupancy. Although both facets and ViCLs can model multiple "live" data for the same memory location, they are inherently different. First, facets are state variables that are updated according to instructions in ILAs and MCM axioms; they are not created or destroyed. In contrast, ViCLs have creation and expiration events in happens-before graphs. Second, facets are more general than ViCLs and are not necessarily tied to caches or other microarchitectural structures. Third, facets enable integration of axiomatic MCMs with operational instruction semantics, while the latter are ignored by ViCLs.

2) Facets vs. Views: In recent work [34, 40], a *view* abstraction was proposed to model the C11 MCM. Our facets are different from views as follows: (i) a view is a map from locations to timestamps, whereas facets are auxiliary state variables, (ii) the views assign explicit timestamps to events, whereas facet-axioms associate events with symbolic timestamps, whose values are not fixed but explored implicitly during verification, (iii) unlike views, facets have been applied in automated SMT-based reasoning.

VI. Conclusions

In this paper, we have presented the ILA-MCM framework, which combines the benefits of operational ILA models with axiomatic MCMs for reasoning about concurrent interactions between heterogeneous components in an SoC. We have introduced a novel facet abstraction that models consistency effects on program-visible states, and use facet-axioms to specify consistency ordering requirements. This provides a constraint-based integration between operational ILA models and axiomatic MCM specifications. Our SMT-based verification procedure supports symbolic reasoning for expressive properties involving both rich instruction semantics and orderings. We have demonstrated two verification applications of our prototype ILA-MCM framework, where we reasoned about an SoC firmware program and a GPU hardware implementation, respectively. Our support for expressive properties allowed synthesizing a malicious exploit in the first case, and our instruction-centric approach enabled compositional verification in the second.

REFERENCES

[1] C. Pilato, Q. Xu, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "On the Design of Scalable and Reusable Accelerators for Big Data Applications," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2016, pp. 406–411.

[2] I. Ohmura, G. Morimoto, Y. Ohno, A. Hasegawa, and M. Taiji, "MDGRAPE-4: a Special-Purpose Computer System for Molecular Dynamics Simulations," *Philosophical Transactions, Series A*, vol. 372, no. 2021, 2014.

[3] J. Rott, "Intel Advanced Encryption Standard Instructions (AES-NI)," *Technical Report, Intel*, 2012.

[4] J. Alglave and M. Tautschnig, "Herding Cats: Modelling, Simulation, Testing, and Data-Mining for Weak Memory," *ACM Transactions on Programming Languages and Systems*, vol. 36, no. 2, pp. 1–7, 2014.

[5] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, "Automatically Comparing Memory Consistency Models," in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2016.

[6] J. Bornholt and E. Torlak, "Synthesizing Memory Models from Framework Sketches and Litmus Tests," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2017, pp. 467–481.

[7] D. Lustig, M. Pellauer, and M. Martonosi, "PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2015, pp. 635–646.

[8] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "CCICheck: Using hb Graphs to Verify the Coherence-Consistency Interface," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2015, pp. 26–37.

[9] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, "COATCheck : Verifying Memory Ordering at the Hardware-OS Interface," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, no. 212, 2016, pp. 233–247.

[10] 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 *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 119–133.

[11] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer, "RTLCheck : Verifying the Memory Consistency of RTL Designs," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2017, pp. 463–476.

[12] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification," 2018. [Online]. Available: http://arxiv.org/abs/1801.01114

[13] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik, "Template-based Synthesis of Instruction-Level Abstractions for SoC Verification," in *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2017, pp. 160–167.

[14] P. Subramanyan, B.-Y. Huang, Y. Vizel, A. Gupta, and S. Malik, "Template-based Parameterized Synthesis of Uniform Instruction-Level Abstractions for SoC Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, no. 99, 2017.

[15] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2013, pp. 1–8.

[16] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[17] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability Modulo Theories," *Handbook of Satisfiability*, vol. 185, pp. 825–885, 2009.

[18] S. Krstic, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor, "Security of SoC Firmware Load Protocols," in *Proceedings of the International Symposium on Hardware-Oriented Security and Trust*, 2014, pp. 70–75.

[19] J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, "Remote-Scope Promotion : Clarified , Rectified , and Verified," in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[20] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[21] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Progranm," *IEEE Transactions on Computers*, no. 9, pp. 690–691, 1979.

[22] Intel, "Intel® 64 and IA-32 Architectures Software Developers Manual," *Volume 3A: System Programming Guide, Chapter 8: Multiple-Processor Management*, no. 253665-067US, 2018.

[23] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by Sketching for Bit-Streaming Programs," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2005, pp. 281–294.

[24] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial Sketching for Finite Programs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 404–415.

[25] J. Lee, D. A. Padua, and S. P. Midkiff, "Basic Compiler Algorithms for Parallel Programs," in *Symposium on Principles and Practice of Parallel Programming*, 1999, pp. 1–12.

[26] C. Wang, S. Kundu, R. Limaye, M. Ganai, and A. Gupta, "Symbolic Predictive Analysis for Concurrent Programs," *Formal Aspects of Computing*, vol. 23, no. 6, pp. 781–805, 2011.

[27] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.

[28] L. Howes and A. Munsh, "The OpenCL Specification," 2015. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf

[29] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-Race-Free Memory Models," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 427–440.

[30] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: a platform for high-level parametric hardware specification and its modular verification," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 1, no. ICFP, pp. 24:1–24:30, 2017.

[31] M. Vijayaraghavan, A. Chlipala, Arvind, and N. Dave, "Modular Deductive Verification of Multiprocessor Hardware Designs," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2015, pp. 109–127.

[32] A. Reid, "Trustworthy Specifications of ARM® v8-A and v8-M System Level Architecture," in *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2017, pp. 161–168.

[33] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, "End-to-End Verification of ARM® Processors with ISA-

Formal," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2016, pp. 42–58.
[34] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer, "A Promising Semantics for Relaxed-Memory Concurrency," in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2017, pp. 175–189.
[35] O. Lahav and D. Dreyer, "Repairing Sequential Consistency in C/C++11," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2016.
[36] O. Lahav, N. Giannarakis, and V. Vafeiadis, "Taming Release-Acquire Consistency," in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2016, pp. 649–662.
[37] V. Vafeiadis and C. Narayan, "Relaxed Separation Logic: A Program Logic for C11 Concurrency," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013, pp. 867–884.
[38] A. Turon, V. Vafeiadis, and D. Dreyer, "GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2014, pp. 691–707.
[39] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning," in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2015, pp. 637–650.
[40] J.-O. Kaiser, H.-H. Dang, D. Dreyer, and O. Lahav, "Strong Logic for Weak Memory : Reasoning About Release-Acquire Consistency in Iris," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2017, pp. 1–17.
[41] A. J. Summers and P. Müller, "Automating Deductive Verification for Weak-Memory Programs," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2018, pp. 190–209.

APPENDIX
FACET-AXIOMS FOR SC AND TSO

The facet-axioms for SC and TSO are shown in Figure 9 and Figure 10, respectively. In the SC model, all facet read/write events are synchronous with the instructions (lines 7-10 in Figure 9), while in TSO model, the two write-facet events of a store instruction follow the program order of the stores (lines 7-13 in Figure 10). Read-facet events are still synchronous (lines 14-15). Fences ensure that previous writes are globally visible at that point, and read-modify-write (RMW) is atomic in the sense that its read and write facets are not breakable (lines 17-21). We define additional functions to specify the corresponding read-from, from-read, and coherence-order relations based on happens-before (HB) relations over facet events, e.g., lines 13-15 in Figure 9 and lines 23-31 in Figure 10. These functions are defined for use in the first axiom in both models.

```
1  Axiom RF_CO_FR
2  forall r:READ | exists w:WRITE |
3     SameAddress[w,r] /\ SameData[w,r]/\ w.decode /\
4        RF[w,r] /\( forall w2:WRITE (not w) |
5           ( SameAddress[w,w2] /\ w2.decode )=>
6              CO[w2,w] \/ FR[r,w2] )
7  Axiom SC_WriteFacetOrder
8  forall w:WRITE | Sync[ w , w.wfe.global ]
9  Axiom SC_ReadFacetOrder
10 forall r: READ | Sync[ r , r.rfe.global ]
11
12 Define RF[w,r] := HB[ w.wfe.global , r.rfe.global ]
13 Define FR[r,w] := HB[ r.rfe.global , w.wfe.global ]
14 Define CO[w1,w2] := HB[ w1.wfe.global , w2.wfe.global ]
```

Fig. 9. Facet-Axioms for SC

```
1  Axiom RF_CO_FR
2  forall r:READ | exists w:WRITE |
3     SameAddress[w,r] /\ SameData[w,r]/\ w.decode /\
4        RF[w,r] /\( forall w2:WRITE (not w) |
5           ( SameAddress[w,w2] /\ w2.decode )=>
6              CO[w2,w] \/ FR[r,w2] )
7  Axiom TSO_WriteFacetOrder
8  forall w:WRITE | HB[ w , w.wfe.local ]  /\
9     HB[ w.wfe.local , w.wfe.global ]
10 Axiom TSO_Store
11 forall w1:WRITE | forall w2: WRITE (not w1) |
12    PO[ w1, w2 ] => HB[ w1.wfe.local, w2.wfe.local]
13       /\ HB[ w1.wfe.global, w2.wfe.global]
14 Axiom TSO_ReadFacetOrder
15 forall r:READ | Sync[ r , r.rfe.local ]
16
17 Axiom TSO_Fence
18 forall f:FENCE | forall w: WRITE | PO[w,f] =>
19    HB[ w.wfe.global, f ]
20 Axiom TSO_RMW
21 forall i:RMW |
22    Sync[i.rfe.local, i.wfe.local, i.wfe.global]
23 Define RF[w,r] :=
24    SameCore[w,r] => HB[w.wfe.local , r.rfe.local ] /\
25   ~SameCore[w,r] => HB[w.wfe.global, r.rfe.local ]
26 Define FR[r,w] :=
27    SameCore[w,r] => HB[r.rfe.local , w.wfe.local ] /\
28   ~SameCore[w,r] => HB[r.rfe.local, w.wfe.global ]
29 Define CO[w1,w2] :=
30    SameCore[w1,w2] => HB[w1.wfe.local, w2.wfe.local] /\
31   ~SameCore[w1,w2] => HB[w1.wfe.global, w2.wfe.global]
```

Fig. 10. Facet-Axioms for TSO