A Formal Account of the Wasm 3.0 Concurrency Model

AZALEA RAAD, Imperial College London, UK
MICHALIS KOKOLOGIANNAKIS, ETH Zurich, Switzerland
VIKTOR VAFEIADIS, MPI-SWS, Germany
CONRAD WATT, Nanyang Technological University, Singapore

WebAssembly (Wasm) is a platform-independent language that serves as a target for web applications and provides rudimentary support for untyped concurrent programming. The Wasm 1.0 memory model was a simple buffer of raw bytes, while the recently finalised Wasm 3.0 adds a new instruction set for dynamically allocated typed structs whose lifetime is managed automatically by the Wasm runtime. However, Wasm structs cannot be used with existing concurrency features of Wasm, and the Wasm type system prevents them from being shared between threads. This means that languages such as Java, OCaml and Kotlin, whose compilers use Wasm 3.0 features, must currently completely forbid the use of source-level shared-memory concurrency features when compiling to Wasm.

As of now, a broad industrial project within the Wasm community named *shared-everything threads* seeks to relax these restrictions and specify the concurrent behaviour of Wasm 3.0 structs. To inform these efforts, we formalise a concurrency semantics for Wasm 3.0 structs and prove the correctness of (a) the intended compilation scheme to x86 and Arm; (b) compilation from C/C++ and OCaml concurrency primitives to Wasm; and (c) intended compiler optimisations. We also establish a DRF property and provide a model checking tool for verifying concurrent Wasm programs. We have carried out our work in close collaboration with members of *WebAssembly's industrial standards community*, with the aim that our semantics should be adopted as the official concurrency model for Wasm 3.0 as the *shared-everything threads* project progresses. Notably, prototypes based on our Wasm 3.0 model are under active implementation by major industrial stakeholders. Along the way, we critically appraise the existing Wasm 1.0 memory model, identifying several changes that could be made to better align it with the state of the art in relaxed memory research.

CCS Concepts: • **Theory of computation → Concurrency**; **Axiomatic semantics**; • **Software and its engineering → Semantics**; *General programming languages*.

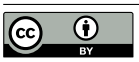
Additional Key Words and Phrases: WebAssembly, Wasm, Memory Models, Memory Consistency, Concurrency

ACM Reference Format:
Azalea Raad, Michalis Kokologiannakis, Viktor Vafeiadis, and Conrad Watt. 2026. A Formal Account of the Wasm 3.0 Concurrency Model. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 333 (October 2026), 29 pages. https://doi.org/10.1145/3839465

1 Introduction

WebAssembly (Wasm) [24] is a low-level virtual machine and bytecode language, supported by all major web browsers and defined through an open standard by the World Wide Web Consortium (W3C). It is intended to be an efficient compilation target for a variety of source languages. Beyond

Authors' Contact Information: Azalea Raad, Imperial College London, London, UK, azalea.raad@imperial.ac.uk; Michalis Kokologiannakis, ETH Zurich, Zurich, Switzerland, michalis.kokologiannakis@inf.ethz.ch; Viktor Vafeiadis, MPI-SWS, Kaiserslautern, Germany, viktor@mpi-sws.org; Conrad Watt, Nanyang Technological University, Singapore, Singapore, conrad.watt@ntu.edu.sg.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/10-ART333
https://doi.org/10.1145/3839465

the web, Wasm has also been used as a platform for software fault isolation [53], in embedded systems [78] and in cloud computing platforms [32].

Wasm 1.0 has been extended with built-in support for multi-threading via statically allocated unstructured memory regions [81]. This mechanism is, however, suboptimal for managed concurrent programming languages such as Java, because it forgoes type safety and requires explicit memory management. Moreover, Wasm 1.0 lacks support for the various flavours of C/C++ atomic memory access modes, thereby requiring implementations to map all C/C++ atomic modes to sequentially consistent (sc) Wasm accesses, incurring an additional performance penalty.

In response, Wasm 3.0 introduces a new set of primitives for allocating and accessing structured memory objects with implicit memory reclamation – the WasmGC extension [64]. Compiling Java code to Wasm 3.0 was shown to double the performance of Google Sheets [72] and improve the performance of web-compiled OCaml code by up to 8 times [54]. However, these objects can only be accessed in the thread in which they were initially allocated, meaning the Wasm 3.0 instruction set can only be targeted when compiling *sequential* source code without any concurrency features. This restriction stems from security and implementation concerns about concurrent garbage collection, especially in relation to historical engineering assumptions made by browser garbage collectors that cross-thread reference cycles are not possible [79]. This severe limitation on the expressivity of WasmGC has limited its adoption; as such, the industrial Wasm community has proposed the *shared* WasmGC extension, currently in development as part of the *shared-everything threads* umbrella standards project [82], which will allow the WasmGC managed objects of Wasm 3.0 to be fully shareable across threads. While this extension is considered essential for Wasm's continued development and adoption, interactions between multi-threading and structured objects are poorly understood in the Wasm community, and therefore academic input is essential to build confidence in the feasibility and correctness of the proposed concurrency extensions.

This paper fills this gap by providing a formal definition of the concurrency model for Wasm 3.0 extended with shared WasmGC, along with an efficient model checker called Waver (WebAssembly VERifier) that allows Wasm developers both to test their code and to explore the shared WasmGC concurrency semantics. We have undertaken all our efforts in close collaboration with members of *WebAssembly's industrial standards community* with the aim of practical adoption. Importantly, prototype extensions to Wasm based on our Wasm 3.0 model are under active implementation in a major Wasm compiler [73]. In carrying out this work, we encountered two major challenges.

First, the Wasm 1.0 concurrency model is defined using a *total order* over all memory accesses, intuitively corresponding to their execution order. This definition is unsuitable for testing/verification purposes: existing tools for generating all possible outcomes of a concurrent Wasm program can only handle tiny programs with straightline code and fewer than 10 memory accesses [51, 80].

Second, our formalisation needs to capture the key property of Wasm 3.0: *unconditional initialisation safety*—that programs should never witness uninitialised objects, which would break type safety. Ensuring this property on weak architectures, such as Armv8, is far from straightforward: it relies crucially on programs not being able to guess the address of a dynamically allocated object.

We address these challenges by developing an *equivalent* formulation of Wasm 1.0 that uses a *partial order* over atomic writes to the same location. We then use this as our basis for modelling the Wasm 3.0 features. To ensure initialisation safety, we record markers for the returns of object constructors within executions, along with an axiom preventing uninitialised reads. We validate our model by proving that it satisfies all expected formal properties (DRF theorem, correctness of compilation and of compiler transformations) and running Waver on several litmus test programs.

```
(module $Wasm1 (memory 1 shared) ;; MP in Wasm 1.0
  (global $x i32 (i32.const 0))          ;; address x
  (global $y i32 (i32.const 0))          ;; address y
  (func $thread1
    (i32.store        (global.get $x) (i32.const 1))
    (i32.atomic.store (global.get $y) (i32.const 1)))
  (func $thread2 (local $a i32) (local $b i32)
    (local.set $a (i32.atomic.load (global.get $y)))
    (local.set $b (i32.load        (global.get $x)))))
```

```
(module $Wasm3                           ;; MP in Wasm 3.0
  (type $t (struct shared (field $x i32) (field $y i32)))
  (global $obj (ref $t) (struct.new_default $t))   ;; {x, y}
  (func $thread1
    (struct.set        $t $x (global.get $obj) (i32.const 1))
    (struct.atomic.set $t $y (global.get $obj) (i32.const 1)))
  (func $thread2 (local $a i32) (local $b i32)
    (local.set $a (struct.atomic.get $t $y (global.get $obj)))
    (local.set $b (struct.get        $t $x (global.get $obj)))))
```

Fig. 1. Wasm 1.0 and Wasm 3.0 code snippets implementing the MP example.

Contributions and Outline. Our contributions, described intuitively in §2, are as follows.

(§3) We reformulate the Wasm 1.0 model in an equivalent fashion that is suitable for automated verification techniques such as model checking, and propose strengthenings that remove certain very weak behaviours without any performance penalty.
(§4) We define a formal concurrency model accounting for Wasm 3.0's dynamically allocated, garbage-collected objects and its proposed weakly consistent C11-style atomics.
(§5) We establish correctness of compilation from OCaml to Wasm, between C/C++ and Wasm (in both directions) and of the default compilation schemes to x86-TSO and Armv8 architectures.
(§6) We prove correctness of all standard compiler transformations.
(§7) We show that strengthening our model to restrict load-store reordering restores DRF.
(§8) We develop Waver, an exploration-optimal, sound and complete model checker for Wasm 3.0 that incorporates state-of-the-art dynamic partial order reduction technique [36] and scales to executions with hundreds of events.

Additional Material. The proofs of all theorems stated in the paper are given in the extended version [61]. Our prototype Waver implementation is available on our project page [62].

2 Overview

We begin with an intuitive account of our contributions. We review the existing Wasm 1.0 concurrency model (§2.1); we then describe the key new feature introduced with Wasm 3.0 along with the challenges it poses when defining its concurrency semantics and a summary of our results (§2.2).

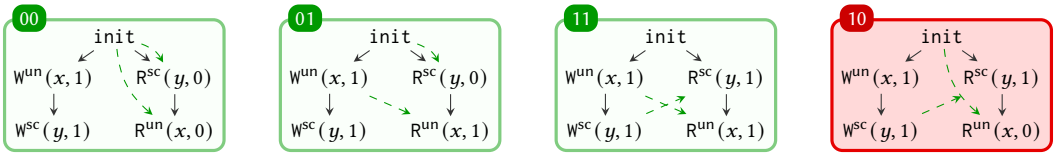
2.1 The Wasm 1.0 Concurrency Model

Wasm 1.0 [81] contains rudimentary support for shared-memory concurrency by providing sequentially consistent accesses, sc, for synchronisation between threads; and unordered accesses, un, for local computation. Watt et al. [80] define the semantics of Wasm 1.0 programs in a declarative (a.k.a. axiomatic) style, where each allowed program execution is represented as a graph that satisfies certain constraints (axioms). The graph nodes record (memory) accesses of the execution, while its edges record various relations among them: (1) the *program order*, po, describes whether an access precedes another in the program's control flow; (2) the *reads-from* relation, rf, associates each read r to the write from which r read its value; and (3) a *total order*, tot, over all accesses.

As an example, consider the "message passing" (MP) program below. As the concrete Wasm syntax is rather verbose (see Fig. 1, left), we use pseudocode throughout. We write $a, b, \ldots$ for thread-local variables and x, y, z for shared memory locations, initialised to 0.

$$\begin{array}{r||l} \tau_1\colon x :=_{\mathsf{un}} 1 & \tau_2\colon a :=_{\mathsf{sc}} y \\ \quad y :=_{\mathsf{sc}} 1 & \quad\ b :=_{\mathsf{un}} x \end{array} \qquad \text{(MP)}$$

Ignoring the tot order, this program has four candidate execution graphs:



The first three executions are *consistent* (allowed by Wasm) and correspond to the outcomes $a = b = 0$, $a = 0 \wedge b = 1$ and $a = b = 1$, respectively. By contrast, Wasm disallows the outcome $a = 1 \wedge b = 0$ depicted in the fourth execution graph 10: Wasm requires that threads not read a stale value if they have observed a more recent one. Formally, this is done by introducing two derived relations: (1) the *synchronised-with* relation, $\mathsf{sw} \triangleq \mathsf{rf} \cap (\mathtt{W}^{\mathsf{sc}} \times \mathtt{R}^{\mathsf{sc}})$, ordering a sc-write before sc-reads that read from it; and (2) the *happens-before* order, $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{sw})^+$, which relates events if they are connected by a sequence of po and sw edges. The said constraint can then be expressed as shown below (left), where $[\mathtt{W}]$ denotes the identity relation on the set of write events ($\mathtt{W}$) and $\mathsf{hb}_{\mathsf{loc}}$ restricts the hb relation to accesses of the same location:

$$\mathsf{rf} \cap (\mathsf{hb}_{\mathsf{loc}}; [\mathtt{W}]; \mathsf{hb}_{\mathsf{loc}}) = \emptyset \qquad \mathtt{W}(x,0) \xrightarrow{\mathsf{hb}} \mathtt{W}(x,1) \xrightarrow{\mathsf{hb}} \mathtt{R}(x,0), \ \mathtt{W}(x,1) \xrightarrow{\mathsf{rf}} \mathtt{R}(x,0)$$

This axiom ensures that a read r from a location x reads from the hb-latest write on x and not a stale write, thereby ruling out scenarios such as that depicted above (right).

The Wasm 1.0 model was designed with two goals: (1) to provide semantics to *all* programs and not deem programs with e.g. data races erroneous, as C/C++ does; and (2) to enable efficient implementations. To support the latter, the Wasm 1.0 constraints are purposely very weak and allow the outcome below, where $/\!/\, v$ denotes the value read by the corresponding load.

$$\begin{array}{l|l} \tau_1\colon x :=_{\mathsf{un}} 1 & \tau_2\colon a :=_{\mathsf{un}} x \,/\!/\, 1 \\ & \quad b :=_{\mathsf{un}} x \,/\!/\, 0 \end{array} \qquad \text{(CoRR)}$$

This behaviour is precluded by *read-read coherence* and is forbidden by all major hardware memory models. It is, however, allowed by the Java memory model [49] to allow compilers to optimise repeated loads even across potential aliasing reads of the same location, enabling the *common sub-expression elimination* transformation below:

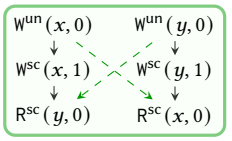
$$\begin{array}{l} a :=_{\mathsf{un}} x \\ b :=_{\mathsf{un}} \mathit{alias\text{-}to\text{-}x} \\ c :=_{\mathsf{un}} x \end{array} \quad \rightsquigarrow \quad \begin{array}{l} a :=_{\mathsf{un}} x \\ b :=_{\mathsf{un}} \mathit{alias\text{-}to\text{-}x} \\ c := a \end{array} \qquad \text{(SubExp-Elim)}$$

Watt et al. [80] have validated the correctness of compilation to x86-TSO and Armv8 architectures through the intermediate memory model (IMM) [57]. However, they have not shown the correctness of compiler transformations such as the above. Proving these transformations correct is important because they are the main reason why Wasm did not simply adopt the C++ model for *relaxed* accesses: the SubExp-Elim optimisation is invalid for C++ relaxed accesses.

Problems with the Wasm 1.0 *Definition.* The Wasm 1.0 definition has two main problems. First, its use of the total order, tot, over *all* memory accesses leads to an enormous state space, rendering the model unusable for model checking—whether stateful, stateless, or SMT-based. We observe, however, that there is no fundamental reason why the Wasm 1.0 semantics need to be defined in such a way. In §3, we show that Wasm 1.0 consistency can be equivalently expressed in terms of a much smaller *modification (or coherence) order* (mo) over all write accesses to the same location, which is uniquely determined in the case of programs without write-write races.

Second, Wasm 1.0 allows some odd weak behaviours that offer no benefit in terms of enabling program transformations and only complicate reasoning about racy programs. An example of such weak behaviour is shown in the following variant of the "store buffering" (SB) litmus test:

$$\begin{array}{l|l} \tau_1\colon x :=_{\mathsf{un}} 0 & \tau_2\colon y :=_{\mathsf{un}} 0 \\ \quad x :=_{\mathsf{sc}} 1 & \quad y :=_{\mathsf{sc}} 1 \\ \quad a :=_{\mathsf{sc}} y \mathbin{/\!/} 0 & \quad b :=_{\mathsf{sc}} x \mathbin{/\!/} 0 \end{array}$$



(SB-alt)

Here, the Wasm 1.0 model allows the weak outcome $a = b = 0$ with the depicted execution graph. Somewhat surprisingly, in this execution the total order tot can be any arbitrary interleaving of the events of the two threads: it does not constrain the reads-from relation at all! We discuss this and other similar oddities of the Wasm 1.0 model in detail in §3.

2.2 Garbage-Collected Objects in Wasm 3.0

Wasm 1.0 was primarily designed to be a compilation target for languages without complex memory management in their runtime, e.g. C, C++, and Rust. Compiling managed languages to Wasm 1.0 is challenging as one must implement their *garbage collector* (GC) programmatically. This is especially onerous since the Wasm-level call stack is not directly accessible by user-level code, and thus tracing GCs can only be implemented with a *shadow stack* simulating the state of the source-level call stack in Wasm's linear memory, resulting in slow, labour-intensive, and incomplete GC implementations.

To address this, Wasm 3.0 introduces new instructions for allocating and accessing garbage-collected *struct types* whose lifetime is managed automatically by the Wasm runtime [64]. Struct types take the form of a list of field types with a statically-checked iso-recursive typing discipline.[1] Struct types are statically declared in the preamble of the Wasm module; instances of a declared type can be dynamically allocated with the struct.new instruction, and their fields can be accessed with struct.get/set instructions. The runtime automatically collects allocated structs once no more live references to them exist. Unlike the existing linear memory of Wasm 1.0, there is no concept of mixed-size or partially overlapping accesses – all struct accesses are directly to one of the declared typed fields. As an example, Fig. 1 (right) shows a version of (MP) with the Wasm 3.0 features.

Structs were designed to reuse parts of the JavaScript GC which is commonly present alongside the Wasm runtime (since both languages coexist in web browser implementations). Due to historical constraints inherited from this platform [79], the initial design for GC structs prevented them by construction from being accessed cross-thread. However, a new effort within the Wasm standards body (*shared-everything threads* [82]) aims to relax this restriction, and therefore it is time to define a suitable concurrency model for the Wasm 3.0 feature-set. Doing so is by no means easy.

Initialisation Safety. A first challenge is *initialisation safety* in the presence of unsynchronised accesses. Ordinarily, when a thread τ_1 allocates a struct, it initialises its fields, and only thereafter (a reference to) the struct might become known to another thread τ_2, which may access its fields. We therefore should ensure that such accesses by τ_2 observe a fully initialised version of the struct, even if τ_2 uses only 'unordered' accesses and thus does not synchronise with τ_1.

$$\begin{array}{l|l} \tau_1\colon a := \mathsf{struct.new}(\{.f :=_{\mathsf{un}} 1, .g :=_{\mathsf{un}} 2\}) & \tau_2\colon b :=_{\mathsf{un}} x \\ \quad x :=_{\mathsf{un}} a & \quad \mathbf{if}(b) \\ & \qquad c :=_{\mathsf{un}} b.f \end{array}$$

Enforcing this property is cheap on modern architectures (it bears no cost on x86-TSO and a mere store-store fence at the end of each constructor on Arm and Power). Formalising it, however, is not

[1] The precise details of Wasm 3.0's type system [64] are not relevant to the concurrency behaviour discussed in this paper.

straightforward as it depends crucially on type safety and on the preservation of data and address dependencies on Armv8 and Power. Since Wasm threads cannot 'guess' the address at which a struct is allocated, every struct access has a certain data flow dependency from the thread that allocated it. Moreover, since the struct address is not leaked during its initialisation, the dependency back to the thread that allocated the struct must start after its initialisation, at which point the store fence along with the dependency chain guarantee the desired property.

Integrating SC with Release/Acquire Accesses. A second challenge that arises when extending the Wasm concurrency model with synchronising non-sc accesses (i.e. C11-style release and acquire accesses) is to combine them correctly. This combination has a long and chequered history. It was first introduced in the 2011 C++ standard, which provided overly strong semantics that implementations on Arm and Power failed to ensure. Lahav et al. [42] observed the problem and corrected the model. The corrections found their way into the 2017 version of the C++ standard, but due to a misunderstanding led to another error, again providing stronger semantics than the implementations guaranteed[2]. This problem was noted only recently and is being fixed.

The key issue is that to allow sc and release/acquire accesses to the same location, one has to distinguish between two different kinds of happens-before paths:

- *happens before*, hb, comprising program-order and synchronisation edges, including synchronisations due to an acquire (or stronger) read reading from a release (or stronger) write;
- *strong happens before*, shb $\triangleq$ po $\cup$ po; hb; po, which is a happens-before path that begins and ends with a program order edge.

We use the former for the coherence axiom that constrains the order of accesses to a single location, and the latter for the SC axiom that constrains the total order in which sc accesses are evaluated. Interestingly, in some corner cases, sc accesses may appear to execute in an order contradicting hb, as the following example (adapted from Lahav et al. [42]) demonstrates:

$$\begin{array}{l|l|l} \tau_1\colon x :=_{\mathsf{sc}} 1 & \tau_2\colon b :=_{\mathsf{sc}} \mathsf{fetch\&add}(y,1)\ /\!/\ 1 & \tau_3\colon y :=_{\mathsf{sc}} 3 \\ \quad y :=_{\mathsf{ra}} 1 & \quad c :=_{\mathsf{ra}} y\ /\!/\ 3 & \quad a :=_{\mathsf{sc}} x\ /\!/\ 0 \end{array} \qquad \text{(Z6.U)}$$

The annotated weak behaviour must be allowed to permit the "leading sync" compilation scheme to Arm7 and Power. In the corresponding execution, the fetch&add happens after τ_1, yet it is ordered before τ_3 (because τ_2 later observes $y :=_{\mathsf{sc}} 3$), and τ_3 is ordered before τ_1 (because τ_3 reads 0).

Type Safety and the DRF Principle. A final challenge is to ensure *type safety*, which is a key desirable property of Wasm 3.0. Simply applying the Wasm 1.0 model to structs forgoes type safety because Wasm 1.0 does not restrict speculative behaviours whatsoever. Consider the LB+data example below. According to the Wasm 1.0 model, a and b can read an arbitrary value (not necessarily of the expected type, thereby breaking type safety).

$$\begin{array}{l|l} \tau_1\colon a :=_{\mathsf{un}} x\ /\!/\ 42 & \tau_2\colon b :=_{\mathsf{un}} y\ /\!/\ 42 \\ \quad y :=_{\mathsf{un}} a & \quad x :=_{\mathsf{un}} b \end{array} \qquad \text{(LB+data)} \qquad \begin{array}{l|l} \tau_1\colon a :=_{\mathsf{un}} x\ /\!/\ 1 & \tau_2\colon b :=_{\mathsf{un}} y\ /\!/\ 1 \\ \quad \mathbf{if}\,(a > 0) & \quad \mathbf{if}\,(b > 0) \\ \quad\quad y :=_{\mathsf{un}} 1 & \quad\quad x :=_{\mathsf{un}} 1 \end{array} \qquad \text{(LB+ctrls)}$$

A minimal fix is to constrain the values read and written in executions to be of the correct type. This means that execution graphs will have to record type information, thereby making the concurrency semantics dependent on the type system. This fix, however, does not resolve a closely related problem with the model demonstrated in the LB+ctrls example above. Wasm 1.0 allows the annotated weak behaviour $a = b = 1$, thereby violating a basic design principle of weak memory models known as the *"data race freedom" (DRF) guarantee.* This principle requires programs to

[2]see https://cplusplus.github.io/LWG/issue3941 for more details.

have sequentially consistent (SC) semantics if their SC executions do not contain any data races.[3] Speculative executions similarly hinder automated program verification [37].

Where necessary, we therefore consider a strengthening of our basic Wasm 3.0 model that forbids speculations altogether by requiring $\mathsf{po} \cup \mathsf{rf}$ to be acyclic as part of consistency. Doing so dispenses with the need for recording type information in execution graphs; we obtain type safety by induction on the $(\mathsf{po} \cup \mathsf{rf})^+$ relation. By a similar inductive argument, in §7 we establish the DRF guarantee.

Results about the Wasm 3.0 *Concurrency Model.* Unlike the original Wasm 1.0 concurrency model, for which Watt et al. [80, 81] only established compilation correctness and a weaker version of the DRF property, we establish all the expected results for our Wasm 3.0 concurrency model. Specifically, we show the following:

(§5) The expected compilation schemes of concurrent accesses to x86-TSO and Armv8 architectures are correct (and ensure initialisation safety).
(§5) Wasm 3.0 allows more efficient compilation of C/C++ and OCaml programs than Wasm 1.0.
(§6) The expected local program transformations (deorderings, access mode strengthenings, sequentialisation, merging adjacent accesses) along with the reordering of unordered reads to the same location are sound.
(§7) Strong Wasm 3.0 (i.e. with $\mathsf{po} \cup \mathsf{rf}$ acyclicity) satisfies the DRF principle: programs have only SC semantics whenever all their SC executions contain no data races; and
(§8) Strong Wasm 3.0 is amenable to efficient model checking: it satisfies prefix-closedness and maximal extensibility, which are the prerequisites for applying the state-of-the-art sound, complete, exploration-optimal, and space-efficient TruSt algorithm [36] to Wasm programs.

Based on the these latter results, we implement a fully-automated verification tool called Waver (Web Assembly VERifier), which takes as input Wasm programs, converts them to LLVM bitcode (via C), and model-checks them with a suitable adaptation of the GenMC model checker [38] that implements the TruSt algorithm. Our implementation involved resolving a number of engineering challenges, which we discuss in more detail in §8. We have applied our model checker to small benchmarks and demonstrate that our Wasm 3.0 model produces the expected behaviours.

3 Reformulating and Repairing the Wasm 1.0 Concurrency Model

We present the existing Wasm 1.0 concurrency model formally (§3.1). We then re-express it in an equivalent fashion that is more suitable for model checking and comparisons with other models, and we propose a strengthening of the model that does not affect the compilation (§3.2).

3.1 The Existing Wasm 1.0 Concurrency Model

Events. In the literature of declarative models, the traces of shared memory accesses generated by a program are commonly represented as a set of *executions*, where each execution is a graph comprising: (i) a set of events (graph nodes); and (ii) a number of relations on events (graph edges).

Each event corresponds to the execution of a memory instruction and is a tuple of the form $e{=}\langle n, \tau, l\rangle$, where $n \in \mathbb{N}$ is the (unique) *event identifier*, $\tau \in \mathrm{TID}$ is the *thread identifier*, and $l \in \mathrm{LAB}$ is the event *label*. The event label records certain information about the memory instruction such as its type (e.g. a read, a write, or an update), the location it accesses and the values read/written. Wasm 1.0 labels additionally record the *memory order*: either *unordered* (un) for regular memory

[3] Watt et al. [81] incorrectly claim that Wasm 1.0 has the DRF guarantee; they establish a *weaker* result showing that if all Wasm-consistent program executions have no data races, then the program has SC semantics.

accesses or *sequentially consistent* (sc) for synchronising accesses. This gives rise to a strict total order $\sqsubset$ such that $\mathsf{un} \sqsubset \mathsf{sc}$. We write $\mathsf{m}_1 \sqsubseteq \mathsf{m}_2$ for $\mathsf{m}_1 \sqsubset \mathsf{m}_2 \vee \mathsf{m}_1 = \mathsf{m}_2$.

Definition 1 (Wasm 1.0 events). A Wasm 1.0 *event* $e \in \textsc{Event}$ is a tuple $\langle n, \tau, l\rangle$, where $n \in \mathbb{N}$ is an event identifier, $\tau \in \textsc{Tid}$ is a thread identifier, and $l \in \textsc{Lab}$ is a Wasm 1.0 *label* which may be:

- $\mathsf{R}^{\mathsf{m}}(x, v)$ to denote reading v from location x with memory order $\mathsf{m} \in \{\mathsf{un}, \mathsf{sc}\}$;
- $\mathsf{W}^{\mathsf{m}}(x, v)$ to denote writing v to location x with memory order $\mathsf{m} \in \{\mathsf{un}, \mathsf{sc}\}$;
- $\mathsf{U}^{\mathsf{sc}}(x, v, v')$ to denote a successful RMW reading value v from x and updating it to v'.

The functions $\mathtt{loc}$, $\mathtt{ord}$, $\mathtt{val}_\mathsf{r}$ and $\mathtt{val}_\mathsf{w}$ respectively project the location, the memory order, the read value and the written value of an event, where applicable. We lift the label functions $\mathtt{loc}$, $\mathtt{val}_\mathsf{r}$ and $\mathtt{val}_\mathsf{w}$ to events, and given an event e, we write e.g. $\mathtt{loc}(e)$ for $\mathtt{loc}(\mathtt{lab}(e))$.

The set of *reads*, R, contains all events with label $\mathsf{R}(.,.)$ or $\mathsf{U}(.,.,.)$; the set of *writes*, W, contains all events with label $\mathsf{W}(.,.)$ or $\mathsf{U}(.,.,.)$; and the set of *updates*, U, contains all events with label $\mathsf{U}(.,.,.)$.

Executions. Wasm 1.0 executions record two relations between events: the *program order*, po, which records the order in which instructions are executed in the program's control flow; and the *reads-from relation*, rf, which associates each read to the write from which it reads its value.

Definition 2 (Wasm 1.0 executions). A Wasm 1.0 *execution* is a tuple $G = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}\rangle$, where:

- $\mathsf{E} \subseteq \textsc{Event}$ denotes a set of Wasm 1.0 events.
- $\mathsf{po} \subseteq \mathsf{E} \times \mathsf{E}$ denotes the *program order*, a strict order on events.
- $\mathsf{rf} \subseteq (\mathsf{E} \cap \mathsf{W}) \times (\mathsf{E} \cap \mathsf{R})$ denotes the *'reads-from' relation* between events with matching locations and values; i.e. $(a, b) \in \mathsf{rf} \Rightarrow \mathtt{loc}(a) = \mathtt{loc}(b) \wedge \mathtt{val}_\mathsf{w}(a) = \mathtt{val}_\mathsf{r}(b)$. Moreover, rf is total and functional on its range: every read is related to exactly one write.

From Programs to Executions. The semantics of a Wasm program P is typically defined as a set of consistent executions associated with P and is defined by straightforward induction on the structure of P. This definition is standard and is omitted (see e.g. [60]).

Notation. Given sets of events A, B, a relation $\mathsf{r} \subseteq A \times B$, a location x and a memory order m, we introduce the following notational conventions. We write A_x for $\{a \in A \mid \mathtt{loc}(a){=}x\}$ and A^{m} for $\{e \in A \mid \mathtt{ord}(e) = \mathsf{m}\}$, when applicable; e.g. W^{sc} denotes the set of write events with sc ordering, and $\mathsf{W}^{\mathsf{sc}}_x$ denotes W^{sc} events on location x. We write $A^{\sqsupseteq \mathsf{m}}$ for $\{e \in A \mid \mathsf{m} \sqsubseteq \mathtt{ord}(e)\}$. We write r_x for $\mathsf{r} \cap (A_x \times B_x)$ and r^{m} for $\mathsf{r} \cap (A^{\mathsf{m}} \times B^{\mathsf{m}})$. We define the *same-location* relation as $\mathsf{sloc} \triangleq \left\{(a, b) \in \textsc{Event} \times \textsc{Event} \;\middle|\; \mathtt{loc}(a) = \mathtt{loc}(b)\right\}$ and write $\mathsf{r}_{\mathsf{loc}}$ for $\mathsf{r} \cap \mathsf{sloc}$. We write $\mathsf{r}^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closures of r, respectively. We write r^{-1} for the inverse of r; $[A]$ for the identity relation on A: $\{(a, a) \mid a \in A\}$; $\mathsf{irefl}(r)$ for $\nexists a.\ (a, a) \in r$; and $\mathsf{acyc}(\mathsf{r})$ for $\mathsf{irefl}(\mathsf{r}^+)$. We write $\mathsf{r}_1; \mathsf{r}_2$ for $\{(a, b) \mid \exists c.\ (a, c) \in \mathsf{r}_1 \wedge (c, b) \in \mathsf{r}_2\}$. When r is a strict partial order, we write $\mathsf{r}|_{\mathsf{imm}}$ for the *immediate* edges in r, i.e. $\mathsf{r}|_{\mathsf{imm}} \triangleq \mathsf{r} \setminus (\mathsf{r}; \mathsf{r})$.

Original Consistency Definition. A memory model's consistency predicate constrains the set of executions to those satisfying certain requirements, e.g. not reading stale or future values. The Wasm 1.0 consistency definition uses two additional relations: tot, a strict total order over all events of the execution, intuitively representing their execution order, and a derived partial order among events known as *happens-before*, hb, which is defined as $\mathsf{hb} \triangleq (\mathsf{po} \cup [\mathsf{E}^{\mathsf{sc}}]; \mathsf{rf}; [\mathsf{E}^{\mathsf{sc}}])^+$ and must agree with the total order tot. Wasm 1.0 further requires rf not to contradict hb: a read cannot read from a future write. Finally, Wasm 1.0 has four axioms preventing reads from reading stale writes according to hb and/or tot depending on whether the corresponding accesses are of sc order.

Definition 3 ([80, Fig. 12]). A Wasm 1.0 execution $G = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}\rangle$ is Wasm 1.0 *consistent* if there exists a strict total order tot on E such that the following hold with $\mathsf{hb} \triangleq (\mathsf{po} \cup [\mathsf{E}^{\mathsf{sc}}]; \mathsf{rf}; [\mathsf{E}^{\mathsf{sc}}])^+$:

- $\mathsf{hb} \subseteq \mathsf{tot}$ — hb respects tot
- $\mathsf{rf} \cap \mathsf{hb}^{-1} = \emptyset$ — Reads cannot read from future writes
- $\mathsf{rf} \cap (\mathsf{hb}_{\mathsf{loc}}; [\mathsf{W}]; \mathsf{hb}_{\mathsf{loc}}) = \emptyset$ — Reads cannot read from stale writes
- $\mathsf{rf} \cap \mathsf{hb} \cap ([\mathsf{W}^{\mathsf{sc}}]; \mathsf{tot}_{\mathsf{loc}}; [\mathsf{W}^{\mathsf{sc}}]; \mathsf{tot}_{\mathsf{loc}}; [\mathsf{R}^{\mathsf{sc}}]) = \emptyset$ — sc reads cannot read from stale sc writes
- $\mathsf{rf} \cap \mathsf{hb} \cap (\mathsf{hb}_{\mathsf{loc}}; [\mathsf{W}^{\mathsf{sc}}]; \mathsf{tot}_{\mathsf{loc}}; [\mathsf{R}^{\mathsf{sc}}]) = \emptyset$ — sc reads cannot read from stale writes
- $\mathsf{rf} \cap \mathsf{hb} \cap ([\mathsf{W}^{\mathsf{sc}}]; \mathsf{tot}_{\mathsf{loc}}; [\mathsf{W}^{\mathsf{sc}}]; \mathsf{hb}_{\mathsf{loc}}) = \emptyset$ — Reads cannot read from stale sc writes

3.2 Reformulating and Strengthening the Wasm 1.0 Model

An Equivalent Definition. Note that in Def. 3 tot is used in the axioms constraining rf in a restricted fashion: it is only used between sc events on the same location. This motivates us to reformulate Def. 3 in a more conventional style that instead uses a *modification order* mo that only relates the sc writes on the same location. We can further rewrite all axioms preventing stale reads into a single axiom by introducing a *reads-before* relation, rb, and requiring it not to contradict hb, along with the standard axiom for sequential consistency on sc accesses. Due to the peculiarities of Wasm 1.0 consistency, the definition of rb is more involved than its conventional definition (i.e. $(\mathsf{rf}^{-1}; \mathsf{mo}) \setminus \mathsf{id}$). This is because stale read constraints also consider non-sc writes to be ordered by hb rather than tot, and rf-edges to be constrained by tot only if they are included in hb.

Definition 4 (mo-consistency). A Wasm 1.0 execution $G = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf} \rangle$ is mo-*consistent* if there exists a strict order mo on E such that:

- $\mathsf{mo} \triangleq \bigcup_{x \in \mathrm{Loc}} \mathsf{mo}_x$, where each mo_x is a strict total order on $\mathsf{W}^{\mathsf{sc}}_x$
- $\mathsf{hb}; \mathsf{eco}$ is irreflexive (partial coherence)
- $[\mathsf{E}^{\mathsf{sc}}]; (\mathsf{hb} \cup \mathsf{mo} \cup \mathsf{rb}); [\mathsf{E}^{\mathsf{sc}}]$ is acyclic (sequential consistency for sc accesses)

where $\mathsf{eco} \triangleq \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb}$ and $\mathsf{rb} \triangleq ((\mathsf{rf} \cap \mathsf{hb})^{-1}; (\mathsf{mo} \cup [\mathsf{W}]; \mathsf{hb}_{\mathsf{loc}}; [\mathsf{W}])) \setminus \mathsf{id}$.

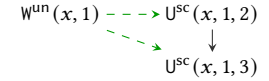
We finally establish that the two characterisations of the Wasm 1.0 model in Def. 3 and Def. 4 are equivalent, with the full proof given in the extended version [61, §A].

Theorem 1 (Equivalence). *An execution G is* Wasm 1.0 *consistent if and only if it is* mo*-consistent.*

Strengthening the Model. Having rewritten the original Wasm 1.0 concurrency model in a more conventional style, we can easily isolate and repair three surprising weaknesses of Wasm 1.0.

First, the definition of rb does not start with rf^{-1} (as its conventional definition does) but rather with $(\mathsf{rf} \cap \mathsf{hb})^{-1}$. This leads to the odd behaviour of (SB-alt) shown in §2, where it is always allowed to read from an unobserved un write, even if it is 'obviously' stale. Moreover, it does not guarantee *atomicity* of RMWs, when reading from a racy un write:

$$\tau_1\colon x :=_{\mathsf{un}} 1 \;\Big\|\; \tau_2\colon \mathbf{if}\,(\mathrm{CAS}^{\mathsf{sc}}(x,1,2))\ \textit{// succeeds} \\ \mathrm{CAS}^{\mathsf{sc}}(x,1,3)\ \textit{// succeeds}$$

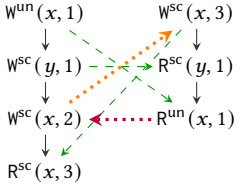


On closer inspection of Def. 3, this is because the stale read axioms only prevent reading from certain hb-earlier writes, but not from an observed racy write.

Second, although the conventional definition of rb composes with mo (i.e. $\mathsf{rb}; \mathsf{mo} \subseteq \mathsf{rb}$), this is not the case for the rb definition in Def. 4. This leads to the following weak behaviour, where the

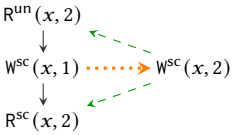
un read of τ_2 reads before the sc-write of τ_1, but not before the mo-later write of τ_2:

$$\begin{array}{l|l} \tau_1\colon x :=_{\mathsf{un}} 1 & \tau_2\colon x :=_{\mathsf{sc}} 3 \\ \quad y :=_{\mathsf{sc}} 1 & \quad b :=_{\mathsf{sc}} y \;/\!/\; 1 \\ \quad x :=_{\mathsf{sc}} 2 & \quad c :=_{\mathsf{un}} x \;/\!/\; 1 \\ \quad a :=_{\mathsf{sc}} x \;/\!/\; 3 & \end{array}$$



Third, Wasm 1.0 consistency allows hb; mo; rf; $[\mathsf{R}^{\mathsf{un}}]$ cycles, which can lead to weak behaviours such as the following, where an un read reads from a future write:

$$\begin{array}{l|l} \tau_1\colon a :=_{\mathsf{un}} x \;/\!/\; 2 & \tau_2\colon x :=_{\mathsf{sc}} 2 \\ \quad x :=_{\mathsf{sc}} 1 & \\ \quad b :=_{\mathsf{sc}} x \;/\!/\; 2 & \end{array}$$



We can rule out these weak behaviours by updating the definitions of rb and eco in Def. 4 as follows:

- $\mathsf{eco} \triangleq \mathsf{rf} \cup \mathsf{mo}; \mathsf{rf}^? \cup \mathsf{rb}$
- $\mathsf{rb} \triangleq (\mathsf{rf}^{-1}; (\mathsf{mo} \cup [\mathsf{W}]; \mathsf{hb}_{\mathsf{loc}}; [\mathsf{W}]; \mathsf{mo}^?)) \setminus \mathsf{id}$

The resulting model is still weaker than the (R)C11 axioms for relaxed and sc accesses, and the strengthening does not affect any of our compilation results. We note that, despite the strengthening, the weak behaviour of CoRR (p. 4) remains allowed and reordering same-location un reads is valid.

A final possible strengthening would be to extend the domain of the modification order to also order un stores and then use the conventional rb definition, i.e. $(\mathsf{rf}^{-1}; \mathsf{mo}) \setminus \mathsf{id}$. Although such a strengthening also does not affect the compilation results, we forgo it because it contradicts the intuitive understanding of 'unordered' memory accesses.

4 Wasm 3.0: Supporting GC Objects, Release/Acquire Accesses and Fences

We present a concurrency model for Wasm 3.0 by extending the Wasm 1.0 model to account for *hitherto unsupported* (1) memory allocation; (2) release/acquire accesses; and (3) fences. Release/acquire accesses are more efficient than their sc counterparts in that they can be compiled with fewer (or cheaper) memory fences, especially on the x86 and Power architectures, and yet provide strong enough guarantees that prohibit the counter-intuitive behaviours exhibited by un accesses.

Definition 5 (Memory orders (revisited)). A *memory order* may be *unordered*, un; *release/acquire*, ra; or *sequentially consistent*, sc; and gives rise to a total order $\sqsubset$, such that $\mathsf{un} \sqsubset \mathsf{ra} \sqsubset \mathsf{sc}$.

We next extend the definition of events with additional labels for fences, allocation and publication of a memory location, which (for simplicity) we associate with a single field of a Wasm struct. The latter two labels are used to give semantics to Wasm 3.0's struct.new construct, which generates a sequence of allocation events (one for each field of the struct), followed by a sequence of write events and a sequence of *publication* events, after which the location(s) may be communicated to other threads. We represent struct.new as a sequence of events (as opposed to a single event) to stress the non-atomic nature of initialisation and the care required for its correct implementation. Another possible representation would be to have a single allocation (resp. publication) event for all the fields of a struct, which would mean that allocation (resp. publication) labels would record a (non-empty) set of locations instead of a single location.

Definition 6 (Wasm 3.0 events). A Wasm 3.0 *event* $e \in \textsc{Event}$ is a tuple $\langle n, \tau, l \rangle$, where $n \in \mathbb{N}$, $\tau \in \textsc{TId}$, and $l \in \textsc{Lab}$ is a *label*, which may be one of the following with $\mathsf{m} \in \textsc{MOrd}$:

- $\mathsf{R}^{\mathsf{m}}(x, v)$ to denote reading v from location x with memory order m;
- $\mathsf{W}^{\mathsf{m}}(x, v)$ to denote writing v to location x with memory order m;
- $\mathsf{U}^{\mathsf{m}}(x, v, v')$ to denote an RMW modifying x to v' when its value matches v, with $\mathsf{m} \in \{\mathsf{ra}, \mathsf{sc}\}$;
- F to denote the execution of a fence;
- $\mathsf{A}(x)$ to denote allocating location x;
- $\mathsf{P}(x)$ to denote making location x public (concluding its initialisation).

Recall that we developed mo-consistency as an alternative characterisation of Wasm-consistency without a total order tot. As discussed, forgoing this total order has several advantages (e.g. for model checking); as such, we develop our extended Wasm model using mo instead. As is standard in the literature of consistency models, hereafter we explicitly record mo as a component of executions.

We further record the *address dependency* (addr) and *data dependency* (data) relations as components of the execution. An address dependency exists when the location accessed by a certain event depends on the value read by a po-earlier read, whereas a data dependency exists when the value written by a write depends on the value read by a po-earlier read. While these dependencies are not used by Wasm 3.0 consistency, they are important for expressing an important *well-formedness* invariant guaranteed by the Wasm syntax: programs cannot 'guess' memory locations. Instead, every location accessed must have already been *published*: it must have been previously allocated either by the thread itself, or by another thread that propagated it to the thread accessing it via a chain of dependencies *after* the location was published. This ensures that the address of a newly allocated object is never published before the end of its constructor.

Definition 7 (Wasm 3.0 executions). A Wasm 3.0 *execution* G is a tuple $\langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{addr}, \mathsf{data} \rangle$ such that E is a set of Wasm 3.0 events, $\langle \mathsf{E}, \mathsf{po}, \mathsf{rf} \rangle$ meets the conditions in Def. 2 and:

- $\mathsf{mo} \triangleq \bigcup_{x \in \textsc{Loc}} \mathsf{mo}_x$ such that each mo_x is a strict total order on $\mathsf{W}_x^{\exists \mathsf{ra}}$.
- $\mathsf{addr} \subseteq [\mathsf{R}]; \mathsf{po}; [\mathsf{R} \cup \mathsf{W}]$ denotes the *address dependency* relation.
- $\mathsf{data} \subseteq [\mathsf{R}]; \mathsf{po}; [\mathsf{W}]$ denotes the *data dependency* relation.

A Wasm 3.0 execution must be *well-formed* in that it must satisfy the following conditions:

- For every event $p \in \mathsf{E} \cap \mathsf{P}$, there exists an event $a \in \mathsf{A}_{\mathsf{loc}(p)}$ such that $a \xrightarrow{\mathsf{po}} p$.
- For every event $e \in \mathsf{E} \cap (\mathsf{R} \cup \mathsf{W})$, there exist events $a \in \mathsf{A}_{\mathsf{loc}(e)}$ and $p \in \mathsf{P}_{\mathsf{loc}(e)}$ such that:
  - $e \in \mathsf{W} \wedge a \xrightarrow{\mathsf{po}} e \xrightarrow{\mathsf{po}} p$ or $a \xrightarrow{\mathsf{po}} p \xrightarrow{\mathsf{po};((\mathsf{rf};\mathsf{data})^*;\mathsf{rf};\mathsf{addr})^?} e$
  - for every e' such that $a \xrightarrow{\mathsf{po}} e' \xrightarrow{\mathsf{po}} p$ it is $e' \in \mathsf{W}$

Execution well-formedness is guaranteed by the mapping of programs to executions.

Extending Wasm-*Consistency.* We extend the notion of mo-consistency (Def. 4) in Def. 8 to account for object allocation, release/acquire accesses, fences and initialisation safety. To this end, we generalise the definition of happens-before to include synchronisation due to release/acquire accesses, incorporate the fixes to rb and eco discussed in §3, change the sc axiom, and introduce three new axioms (all extensions to Wasm 1.0 consistency (Def. 4) are highlighted in Def. 8):

- A successful RMW event must read from its immediate previous write and cannot read from a write older than that, i.e. rb; mo is irreflexive. We remark that this standard axiom was not needed in the Wasm 1.0 definition as it followed from the sc axiom; with release/acquire RMWs, this is no longer the case and we need a separate atomicity axiom. (ATOMICITY)
- Every allocation event returns a fresh abstract location, i.e. one that is not returned by another allocation event (and thus each location is allocated at most once). (ALLOC)

- Reads see fully initialised objects and cannot read before they are published. (INIT-SAFETY)

Concerning the sc axiom, recall that requiring $[\mathsf{E}^{\mathsf{sc}}];(\mathsf{hb}\cup\mathsf{mo}\cup\mathsf{rb});[\mathsf{E}^{\mathsf{sc}}]$ acyclicity prevents efficient compilation on Power and Arm7 [42]. We therefore introduce the *strong happens-before* order, shb, whereby we restrict hb paths to those that start and end with a po edge. We then include shb and rf between sc events in the sc acyclicity axiom. Fences impose a few more orderings (see the psc_F definition), which in essence upgrade the memory order of hb-related accesses to sc.

Definition 8 (Wasm 3.0 consistency). A Wasm 3.0 execution $G = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{addr}, \mathsf{data}\rangle$ is Wasm 3.0 *consistent* if the following hold:

- $\mathsf{hb};\mathsf{eco}$ is irreflexive (COHERENCE)
- $[\mathsf{E}^{\mathsf{sc}}];(\mathsf{shb}\cup\mathsf{rf}\cup\mathsf{mo}\cup\mathsf{rb}\cup\mathsf{psc}_\mathsf{F});[\mathsf{E}^{\mathsf{sc}}]$ is acyclic (SC)
- $\mathsf{rb};\mathsf{mo}$ is irreflexive (ATOMICITY)
- $\forall x \in \mathrm{Loc}.\ |\mathsf{A}_x| \leq 1$ (ALLOC)
- $\mathsf{rb};\mathsf{po}_{\mathsf{loc}};[\mathsf{P}] = \emptyset$ (INIT-SAFETY)

with $\mathsf{hb} \triangleq (\mathsf{po}\cup[\mathsf{W}^{\sqsupseteq\mathsf{ra}}];\mathsf{rf};[\mathsf{R}^{\sqsupseteq\mathsf{ra}}])^+$ $\quad \mathsf{eco} \triangleq \mathsf{rf}\cup\mathsf{mo};\mathsf{rf}^?\cup\mathsf{rb}$

$\mathsf{shb} \triangleq \mathsf{po}\cup(\mathsf{po};\mathsf{hb};\mathsf{po})$ $\quad \mathsf{rb} \triangleq (\mathsf{rf}^{-1};(\mathsf{mo}\cup[\mathsf{W}];\mathsf{hb}_{\mathsf{loc}};[\mathsf{W}];\mathsf{mo}^?)) \setminus \mathsf{id}$

$\mathsf{psc}_\mathsf{F} \triangleq [\mathsf{F}];\mathsf{hb};(\mathsf{mo}\cup\mathsf{rb});[\mathsf{E}^{\mathsf{sc}}]\cup[\mathsf{E}^{\mathsf{sc}}];(\mathsf{mo}\cup\mathsf{rb});\mathsf{hb};[\mathsf{F}]\cup[\mathsf{F}];\mathsf{hb};\mathsf{eco};\mathsf{hb};[\mathsf{F}]$

Alternative Wasm 3.0 *Characterisation.* To keep our presentation of the Wasm 3.0 model close to that of Wasm 1.0, we opted to present RMW instructions as a *single* event. An alternative approach is to present an RMW as a *pair* of events, a read and a write, respectively capturing the 'read' and 'modify-write' phases of the RMW. Both presentations are common in the literature: OCaml [18] and x86-TSO [5] take the first (single-event) approach, Armv8 [58] takes the second, while RC11 uses both approaches to give two equivalent characterisations. As such, to prove the correctness of our compilation to Armv8 (later in §5.2), we do so using an *alternative* Wasm 3.0 presentation that represents RMWs as a pair of events. We then show that the two characterisations of Wasm 3.0 are *equivalent* (see the extended version [61, §C]).

Strong Wasm 3.0 *Consistency.* As discussed in §2, the Wasm 1.0 and Wasm 3.0 models allow unrestricted load buffering behaviours (e.g. LB+ctrls), and thus the DRF guarantee does not directly hold. To this end, we introduce *strong* Wasm *consistency* (Def. 9), additionally prohibiting $\mathsf{po}\cup\mathsf{rf}$ cycles (and thus all load buffering). This allows us to establish our DRF theorem later in §7.

Definition 9 (Strong Wasm 3.0 consistency). A Wasm 3.0 execution $G{=}\langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{addr}, \mathsf{data}\rangle$ is *strongly* Wasm 3.0 *consistent* if (1) G is Wasm 3.0 consistent (Def. 8); and (2) $\mathrm{acyc}(\mathsf{po}\cup\mathsf{rf})$ holds.

5 Compilation Correctness Results

We show that Wasm 3.0 is suitable for a bytecode language by showing that it can both efficiently be compiled to common hardware architectures (x86-TSO and Armv8) and serve as a compilation target for higher-level languages (C/C++ and OCaml). To this end, we prove the correctness of the following compilation schemes (we present the last two results in the appendix for lack of space):

(§5.1) Compilation from Wasm to x86-TSO;
(§5.2) Compilation from Wasm to Armv8;
(§5.3) Compilation from C/C++ to Wasm;
(§5.4) Compilation from OCaml to Wasm; and
(§5.5) Compilation from Wasm to C/C++.

For brevity, we omit the existing models of x86-TSO, Armv8, C/C++ and OCaml; we present them in the accompanying technical appendix, where we accordingly extend them to ensure allocation freshness (we highlight these extensions for clarity).

Compilation Correctness. Compiling a source program P_{src} to a target program P_{trg} is *correct* if every *outcome* obtained by running P_{trg} against the target consistency model T can be obtained by running P_{src} against the source consistency model S. That is, for every T-consistent execution G_{trg} of P_{trg}, either there exists an *unsafe* S-consistent execution of P_{src}, or G_{trg} is *safe* and there exists an S-consistent execution of P_{src} with the same *outcome*.

An execution is *safe* according to a memory model M if it never accesses unallocated memory and does not contain any data races that M forbids. In the case of Wasm, all well-formed executions are *safe*; however, this is not generally the case for other memory models.

What remains is to define the outcome of an execution G. A common approach in the literature is to define the outcome of G as a function O^{mo} that maps each location x to the *last* value written to x in G according to mo. Two executions G_1, G_2 are then said to have the same outcome if $G_1.O^{\text{mo}}(y) = G_2.O^{\text{mo}}(y)$, for all locations y. However, this definition of outcomes is not suitable for Wasm as unordered (un) stores in Wasm do not participate in mo.

To this end, we assume the existence of a designated *outcome location*, $x^{\text{out}} \in \text{Loc}$, that is never accessed in an unordered (un) fashion. We then define G_1 and G_2 to have the same outcome if the last value written to x^{out} (in mo) is the same in both.

Note that this definition is general enough to allow us to distinguish two executions that hold different values (outcomes) for *any* location y. Specifically, let G_{src} be an execution of a given source program P_{src} and let G_{trg} be the corresponding target execution. Let us assume that G_{src} and G_{trg} respectively hold values v_s and v_t for a location y (where $v_s \neq v_t$), and let $P'_{\text{src}} \triangleq P_{\text{src}}; a := y; x^{\text{out}} := a$. That is, we extend the original program to read the value of y and subsequently write it to x^{out}. It is then straightforward to see that P'_{src} results in an execution G'_{src} that contains a write event with value v_s to x^{out}, that G'_{trg} can be obtained from G'_{src} through the same compilation scheme, and that G'_{trg} contains a write event with value v_t to x^{out}. As such, G'_{src} and G'_{trg} have different outcomes.

Definition 10 (Outcomes). Assume a designated location $x^{\text{out}} \in \text{Loc}$ that is never accessed via un order. The *outcome* of an execution G, written $\text{Out}(G)$, denotes the $\text{mo}_{x^{\text{out}}}$-maximal value in G.

Compilation Schemes. As is standard, we define the compilation scheme from a source model S to a target model T as a function, $(\!|.|\!)_{\text{S}\to\text{T}}$ mapping each S-label to a sequence of T-labels. Intuitively, this denotes compiling each S-instruction to a sequence of (sequentially composed) T-instructions.

Definition 11 (Compilation correctness). A compilation scheme $(\!|.|\!)_{\text{S}\to\text{T}}$ is *correct* if for every source program P and every T-consistent execution G_{trg} of $(\!|P|\!)_{\text{S}\to\text{T}}$, there exists an S-consistent execution G_{src} of P such that either G_{src} is not safe (according to S), or G_{trg} is safe (according to T) and $\text{Out}(G_{\text{src}}) = \text{Out}(G_{\text{trg}})$.

Note that, in particular, compilation correctness from Wasm 3.0 to a target architecture implies that all target executions are safe.

5.1 Correct Compilation from Wasm 3.0 to x86-TSO

The x86-TSO [68] is simply referred to as TSO in the literature. For brevity, we do not repeat this model here and instead present it in the extended version [61, §B], where we further extend TSO executions to account for *memory allocation* and ensure allocation freshness. Compared to Wasm and RC11, TSO is a simple model in that TSO memory accesses are not associated with an order.

A TSO execution G is *safe* if for every access of a location x, there exists an allocation of x that precedes it in $G.\mathrm{po}$ or $G.\mathrm{ob}$, where ob is the TSO 'ordered-before' relation and is defined in the extended version [61, §B].

Definition 12 (Safe TSO execution). A TSO execution G is *safe* if for all $e \in G.\mathsf{E} \cap (\mathsf{R} \cup \mathsf{W})$, there exists $a \in \mathsf{A}_{\mathtt{loc}(e)}$ such that $a \xrightarrow{G.\mathrm{po}} e$ or $a \xrightarrow{G.\mathrm{ob}} e$.

TSO *Compilation Scheme.* As TSO is comparatively stronger than Wasm 3.0, it allows us to compile each Wasm 3.0 load (resp. RMW) with order m to a TSO load (resp. RMW), regardless of the memory order m. Similarly, a Wasm un or ra store can be simply compiled to a TSO store. However, as TSO allows store-load reordering, a Wasm sc store is compiled to a TSO store *followed by a memory fence* to prevent such reordering. Finally, as expected, Wasm fences and allocations are compiled to the corresponding TSO instruction, while a Wasm publication is compiled to the empty sequence of labels. Our Wasm-to-TSO compilation scheme is identical to that of C11-to-TSO in the GCC and LLVM compilers, as well as that of RC11-to-TSO [42].

Definition 13 (TSO compilation scheme). The TSO *compilation scheme*, $(\!|.|\!)_{\mathrm{W}\to\mathrm{TSO}} : \textsc{Lab} \to \overline{\textsc{Lab}^{\mathrm{TSO}}}$, is defined as follows, mapping each Wasm label to a sequence of TSO labels:

$$(\!|\mathsf{R}^{\mathsf{m}}(x,v)|\!)_{\mathrm{W}\to\mathrm{TSO}} \triangleq \mathsf{R}(x,v) \qquad (\!|\mathsf{W}^{\mathsf{m}\neq\mathsf{sc}}(x,v)|\!)_{\mathrm{W}\to\mathrm{TSO}} \triangleq \mathsf{W}(x,v) \qquad (\!|\mathsf{W}^{\mathsf{sc}}(x,v)|\!)_{\mathrm{W}\to\mathrm{TSO}} \triangleq \mathsf{W}(x,v);\mathsf{MF}$$

$$(\!|\mathsf{U}^{\mathsf{m}}(x,v,v')|\!)_{\mathrm{W}\to\mathrm{TSO}} \triangleq \mathsf{U}(x,v,v') \qquad (\!|\mathsf{F}|\!)_{\mathrm{W}\to\mathrm{TSO}} \triangleq \mathsf{MF} \qquad (\!|\mathsf{A}(x)|\!)_{\mathrm{W}\to\mathrm{TSO}} \triangleq \mathsf{A}(x) \qquad (\!|\mathsf{P}(x)|\!)_{\mathrm{W}\to\mathrm{TSO}} \triangleq \epsilon$$

Theorem 2 (TSO compilation)**.** *The compilation scheme* $(\!|.|\!)_{\mathrm{W}\to\mathrm{TSO}}$ *from (strong)* Wasm 3.0 *to* TSO *is correct.*

The full proof is given in the extended version [61, §B].

5.2 Correct Compilation from Wasm 3.0 to Armv8

We present the Armv8 model [58] in the extended version [61, §D]; as with TSO, we extend Armv8 executions to account for *memory allocation*. Similar to Wasm 1.0 and Wasm 3.0, Armv8 loads and stores specify an order: an Armv8 load may be *plain* (P), *acquire-SC* (A) or *acquire-PC* (Q); an Armv8 store may be *plain* (P) or *release* (L).[4] Additionally, Armv8 offers a fence instruction, known as a *data memory barrier*, which is also associated with an order that may be *store* (st) or *full* (sy). Intuitively, when inserted between two stores, a store fence prevents them from being reordered. The full fence is stronger and prevents reordering when inserted between *any* two instructions.

Unlike TSO, Armv8 architectures do not include explicit labels for update (RMW) events. Instead, in Armv8 a failed RMW is simply represented as a load event, and a successful RMW event on a location x is captured through two events, a load e_l and a store e_s (respectively capturing the read and modify/write phases of the RMW) on x that are immediately adjacent in the program order. To distinguish a load-store pair (e_l, e_s) associated with a successful RMW from a regular load followed by a regular store, the constituent load and store are related by the *RMW relation* rmw, i.e. $(e_l, e_s) \in \mathrm{rmw}$. As such, Armv8 executions additionally record their associated rmw relation.

Finally, similar to Wasm executions, Armv8 executions record program dependencies through addr, data and ctrl relations to track address, data and control dependencies, respectively.

Armv8 execution safety is analogous to that of TSO, accordingly using the notion of ob in Armv8.

Definition 14 (Armv8 safe executions). An Armv8 execution G is *safe* if if for all $e \in G.\mathsf{E} \cap (\mathsf{R} \cup \mathsf{W})$, there exists $a \in \mathsf{A}_{\mathtt{loc}(e)}$ such that $a \xrightarrow{G.\mathrm{po}} e$ or $a \xrightarrow{G.\mathrm{ob}} e$.

[4] Throughout this paper, we only present the fragment of Armv8 components that are relevant for our compilation scheme.

Armv8 *Compilation Scheme.* We next present our compilation scheme from Wasm to Armv8. As before, we define the compilation scheme as a function mapping each Wasm label to a sequence of Armv8 labels. Specifically, un accesses map to corresponding P (plain) accesses; ra/sc loads map to A loads; and ra/sc stores map to L stores. As discussed, a successful RMW with order m maps to a load with order A followed immediately (in program order) by a store with order L. Finally, a Wasm allocation simply maps to an Armv8 allocation, a Wasm fence maps to a full Armv8 fence, and a Wasm publication maps to a store fence.

Intuitively, mapping the publication of x to a store fence is sufficient to ensure that the resulting program after Armv8 compilation remains safe, i.e. it only accesses allocated locations and only once they are published. Specifically, the store fence ensures that the earlier allocation a and initialisation of x (po-before its publication) remain ordered before all later writes (on any location). As such, since the existence of location x is published to other threads through a write (recall from well-formedness in Def. 7 that for each access e on x in a different thread we have $a \xrightarrow{\mathsf{po};[\mathsf{P}_x];\mathsf{po}} w \xrightarrow{(\mathsf{rf};\mathsf{data})^*;\mathsf{rf};\mathsf{addr}} e$ for some write w), this ensures that the allocation of x and the write publishing it remain ordered.

Definition 15 (Armv8 compilation scheme). The Armv8 *compilation scheme*, $(\!|.|\!)_{\mathrm{W}\to\mathrm{Armv8}} : \textsc{Lab} \to \overline{\textsc{Lab}^{\mathrm{Armv8}}}$, is defined as follows, mapping each Wasm label to a sequence of Armv8 labels:

$$(\!|\mathsf{R}^{\mathsf{m}}(x,v)|\!)_{\mathrm{W}\to\mathrm{Armv8}} \triangleq \mathsf{R}^{\mathsf{rd}(\mathsf{m})}(x,v) \qquad (\!|\mathsf{W}^{\mathsf{m}}(x,v)|\!)_{\mathrm{W}\to\mathrm{Armv8}} \triangleq \mathsf{W}^{\mathsf{wr}(\mathsf{m})}(x,v) \qquad (\!|\mathsf{A}(x)|\!)_{\mathrm{W}\to\mathrm{Armv8}} \triangleq \mathsf{A}(x)$$

$$(\!|\mathsf{U}^{\mathsf{m}}(x,v,v')|\!)_{\mathrm{W}\to\mathrm{Armv8}} \triangleq \mathsf{R}^{\mathsf{A}}(x,v);\mathsf{W}^{\mathsf{L}}(x,v') \qquad (\!|\mathsf{F}|\!)_{\mathrm{W}\to\mathrm{Armv8}} \triangleq \mathsf{DMB}^{\mathsf{sy}} \qquad (\!|\mathsf{P}(x)|\!)_{\mathrm{W}\to\mathrm{Armv8}} \triangleq \mathsf{DMB}^{\mathsf{st}}$$

with $\mathsf{rd}(\mathsf{un}) \triangleq \mathsf{P}$, $\mathsf{rd}(\mathsf{ra}) = \mathsf{rd}(\mathsf{sc}) \triangleq \mathsf{A}$, $\mathsf{wr}(\mathsf{un}) \triangleq \mathsf{P}$ and $\mathsf{wr}(\mathsf{ra}) = \mathsf{wr}(\mathsf{sc}) \triangleq \mathsf{L}$.

Theorem 3 (Armv8 compilation)**.** *The compilation* $(\!|.|\!)_{\mathrm{W}\to\mathrm{Armv8}}$ *from* Wasm 3.0 *to* Armv8 *is correct.*

To prove this theorem, we first show that our alternative Wasm 3.0 model (where we present each RMW via two events, see the extended version [61, §C]) is *equivalent* to the Wasm 3.0 model in Def. 8, and then prove the correctness of our Armv8 compilation scheme against our alternative Wasm 3.0 model. The full proof is given in the extended version [61, §D].

We remark that this compilation scheme is *not* correct for strong Wasm 3.0 consistency because it does not rule out $\mathsf{po} \cup \mathsf{rf}$ cycles. There are several ways to prevent such cycles, e.g. by compiling all Wasm un reads to Armv8 A/Q reads, or by compiling all Wasm un writes to Armv8 L writes. These stronger compilation schemes from strong Wasm 3.0 to Armv8 are trivially correct because they ensure that $[\mathsf{R}];\mathsf{po};[\mathsf{W}] \in \mathsf{bob} \subseteq \mathsf{ob}$, hence also ensuring the absence of $\mathsf{po} \cup \mathsf{rf}$ cycles.

5.3 Correct Compilation from C/C++ to Wasm 3.0

The C/C++ model [9] predates Wasm 1.0 and Wasm 3.0 and has gone through a number of iterations. Here, we opt for a repaired version of the model by Lahav et al. [42], referred to as RC11. RC11 provides more memory orders than Wasm—namely, non-atomic loads and stores (na), relaxed-atomic accesses (rlx), acquire loads, RMWs, and fences (acq), release stores, RMWs, and fences (rel), acquire-release RMWs and fences (acqrel), and sequentially consistent accesses and fences (sc)—but lacks a notion of unordered accesses. The closest RC11 memory orders to Wasm unordered accesses are RC11's non-atomics (which have undefined semantics in case of a data race and are thus strictly weaker than Wasm unordered accesses) and RC11's relaxed-atomic accesses (which are strictly stronger and satisfy a stronger coherence axiom that forbids the weak behaviour of CoRR).

RC11 executions are similar to Wasm executions, except that mo relates *all* writes to the same location, and that addr and data components are not used. RC11 consistency is quite involved and discussed in detail by Lahav et al. [42]. For brevity, we omit the RC11 definition here and repeat it in the extended version [61, §E], where we further extend it to account for memory allocation.

In RC11, a *data race* involving a non-atomic access is considered a program error ('undefined behaviour'), where two accesses constitute a data race if they are on the same location, at least one of them is a write, and they are not related by the (RC11) '*happens-before*' relation (hb). As such, in what follows we define an RC11 execution G to be *safe* RC11 if (1) for every access of a location x, there exists an allocation of x that precedes it in $G.\mathsf{hb}$ (ensuring allocation safety); and (2) G does not exhibit any data races involving non-atomic accesses.

Definition 16. An RC11 execution G is *safe* if the following hold:

- For all $e \in G.\mathsf{E} \cap (\mathsf{R} \cup \mathsf{W})$, there exists $a \in \mathsf{A}_{\mathtt{loc}(e)}$ such that $a \xrightarrow{G.\mathsf{hb}} e$.
- For all $x \in \mathrm{Loc}$ and $a, b \in G.\mathsf{E} \cap (\mathsf{R}_x \cup \mathsf{W}_x)$, if $\mathsf{m}(a) = \mathsf{na}$ and at least one of a and b is a write event (i.e. $a \in \mathsf{W} \vee b \in \mathsf{W}$), then $a = b$ or $a \xrightarrow{G.\mathsf{hb}} b$ or $b \xrightarrow{G.\mathsf{hb}} a$.

Compilation Scheme. Compiling RC11 to Wasm 3.0 simply maps non-atomic accesses to un ones, sc-atomic accesses to (Wasm 3.0) sc ones, and remaining RC11 atomic accesses to ra ones.

$$(\!|\mathsf{m}|\!)_{\mathrm{C}\to\mathrm{W}} \triangleq \begin{cases} \mathsf{un} & \text{if } \mathsf{m} = \mathsf{na} \\ \mathsf{sc} & \text{if } \mathsf{m} = \mathsf{sc} \\ \mathsf{ra} & \text{otherwise} \end{cases} \qquad \begin{array}{ll} (\!|\mathsf{R}^{\mathsf{m}}(x,v)|\!)_{\mathrm{C}\to\mathrm{W}} & \triangleq \mathsf{R}^{(\!|m|\!)_{\mathrm{C}\to\mathrm{W}}}(x,v) \\ (\!|\mathsf{W}^{\mathsf{m}}(x,v)|\!)_{\mathrm{C}\to\mathrm{W}} & \triangleq \mathsf{W}^{(\!|m|\!)_{\mathrm{C}\to\mathrm{W}}}(x,v) \\ (\!|\mathsf{U}^{\mathsf{m}}(x,v,v')|\!)_{\mathrm{C}\to\mathrm{W}} & \triangleq \mathsf{U}^{(\!|m|\!)_{\mathrm{C}\to\mathrm{W}}}(x,v,v') \end{array}$$

$$(\!|\mathsf{F}^{\mathsf{sc}}|\!)_{\mathrm{C}\to\mathrm{W}} \triangleq \mathsf{F}^{\mathsf{sc}} \qquad (\!|\mathsf{F}^{\mathsf{m}\neq\mathsf{sc}}|\!)_{\mathrm{C}\to\mathrm{W}} \triangleq \epsilon \qquad (\!|\mathsf{A}(x)|\!)_{\mathrm{C}\to\mathrm{W}} \triangleq \mathsf{A}(x) \qquad (\!|\mathsf{P}(x)|\!)_{\mathrm{C}\to\mathrm{W}} \triangleq \mathsf{P}(x)$$

It is straightforward to show that this compilation scheme is correct. Since we can assume that the source program has no data races, writes in the target execution graph are always ordered by $\mathsf{hb} \cup \mathsf{mo}_{\mathrm{Wasm}}$, and so we can set $\mathsf{mo}_{\mathrm{RC11}} \triangleq [\mathrm{W}]; \mathsf{hb}_{\mathtt{loc}}; [\mathrm{W}] \cup \mathsf{mo}_{\mathrm{Wasm}}$, and the RC11 axioms follow directly from the corresponding Wasm 3.0 ones.

Theorem 4 (C→W). *The compilation scheme* $(\!|_|\!)_{\mathrm{C}\to\mathrm{W}}$ *from* RC11 *to strong* Wasm 3.0 *is correct.*

Similarly, we can show that $(\!|_|\!)_{\mathrm{C}\to\mathrm{W}}$ is a correct compilation scheme from *weak* RC11 to Wasm 3.0, where weak RC11 is defined exactly like RC11 without the $\mathsf{po} \cup \mathsf{rf}$ acyclicity axiom.

5.4 Correct Compilation from Multicore OCaml to Wasm 3.0

Multicore OCaml [18] has two types of locations: (1) regular non-atomic locations, which can be read and written, and are meant for local computation; and (2) atomic locations, which can be accessed in a sequentially consistent fashion with reads, writes, and atomic RMWs. Unlike C/C++, and similar to Wasm, data races do not have undefined semantics. Racy non-atomic reads simply read from some write, and OCaml-consistency predicate determines when such rf edges are valid.

As in C/C++, OCaml executions contain a modification order, mo, that totally orders all writes to the same location—not only the SC-atomic ones. We repeat the definition of OCaml-consistency in the extended version [61, §E], where we extend it to account for memory allocation. The definition of OCaml-consistency is straightforward but surprisingly strong in two ways. First, its happens-before order includes not only rf-edges between SC accesses, but also mo-edges between them. This effectively means that its SC writes behave exactly as RMWs, and indeed have to be compiled as such. Second, as part of its CAUSALITY axiom, OCaml enforces $\mathsf{po} \cup \mathsf{rf}$ acyclicity even on non-atomic accesses, and so they cannot be compiled to Wasm 3.0 as unordered accesses; they are mapped to ra accesses.

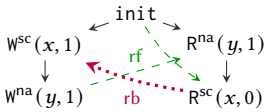
Theorem 5. *The following compilation scheme* $(\!|-|\!)_{\mathrm{O}\to\mathrm{W}}$ *from* OCaml *to* Wasm 3.0 *is correct:*

$$\begin{array}{lll} (\!|\mathsf{R}^{\mathsf{na}}(x,v)|\!)_{\mathrm{O}\to\mathrm{W}} \triangleq \mathsf{R}^{\mathsf{ra}}(x,v) & (\!|\mathsf{R}^{\mathsf{sc}}(x,v)|\!)_{\mathrm{O}\to\mathrm{W}} \triangleq \mathsf{R}^{\mathsf{sc}}(x,v) & (\!|\mathsf{A}(x)|\!)_{\mathrm{O}\to\mathrm{W}} \triangleq \mathsf{A}(x) \\ (\!|\mathsf{W}^{\mathsf{na}}(x,v)|\!)_{\mathrm{O}\to\mathrm{W}} \triangleq \mathsf{W}^{\mathsf{ra}}(x,v) & (\!|\mathsf{W}^{\mathsf{sc}}(x,v)|\!)_{\mathrm{O}\to\mathrm{W}} \triangleq \mathsf{U}^{\mathsf{sc}}(x,_,v) & (\!|\mathsf{U}^{\mathsf{sc}}(x,v,v')|\!)_{\mathrm{O}\to\mathrm{W}} \triangleq \mathsf{U}^{\mathsf{sc}}(x,v,v') \end{array}$$

The proof of this theorem is straightforward. Mapping OCaml's sc writes to RMWs implies that $[\mathsf{E}^{\mathsf{sc}}]; \mathsf{mo}; [\mathsf{E}^{\mathsf{sc}}] \subseteq \mathsf{rf}$, and therefore $\mathsf{hb}_{\mathsf{OCaml}} \subseteq \mathsf{hb}_{\mathsf{Wasm}}$, while mapping OCaml's non-atomic accceses to ra accesses implies that $\mathsf{rf} \subseteq \mathsf{hb}_{\mathsf{Wasm}}$, $\mathsf{mo}_{\mathsf{OCaml}} = \mathsf{mo}_{\mathsf{Wasm}}$ and $\mathsf{rb}_{\mathsf{OCaml}} = \mathsf{rb}_{\mathsf{Wasm}}$. From these facts, the OCaml axioms directly follow from the Wasm 3.0 ones.

Remark. Mapping OCaml's non-atomic accesses to Wasm unordered accesses is incorrect even for *strong* Wasm 3.0 (the strengthening of Wasm 3.0 with $\mathsf{po} \cup \mathsf{rf}$ acyclicity in Def. 9). This is not only because of the stronger coherence requirements of the OCaml model where mo also ranges over non-atomic accesses, but also because OCaml's CAUSALITY axiom is surprisingly strong and forbids the annotated weak behaviour of the following "message passing" variant:

$$\begin{array}{rl} \tau_1\colon x :=_{\mathsf{sc}} 1 & \tau_2\colon a :=_{\mathsf{na}} y \mathbin{/\!/} 1 \\ y :=_{\mathsf{na}} 1 & b :=_{\mathsf{sc}} x \mathbin{/\!/} 0 \end{array}$$



(MP2)

Here, the weak behaviour is allowed by Wasm 1.0 and Wasm 3.0 (if we model na as un) and by RC11 (if we model na as rlx), but forbidden by OCaml. Interestingly, this weak behaviour is also allowed by Armv8 if we map non-atomic accesses to plain Armv8 loads and stores — Multicore OCaml was designed in full knowledge that it would not support such a compilation scheme [18].

Furthermore, note that in the absence of release/acquire atomics from Wasm 3.0 (namely, in the original Wasm 1.0 model), all OCaml accesses must be mapped to sc atomics, incurring an additional performance penalty.

5.5 Correct Compilation from Wasm 3.0 to C/C++

We next consider compilation between C/C++ and Wasm 3.0 in the reverse direction: from Wasm 3.0 to C/C++. Compiling in the reverse direction is useful from both theoretical and practical perspectives. From a theoretical perspective, showing correctness of compilation from Wasm 3.0 to RC11 is an easy way of achieving correctness of compilation to hardware models by appealing to the correctness of compilation from RC11 to x86, Armv8, and Power. From a practical perspective, there is indeed a prominent compiler from Wasm to C, `wasm2c` [23], that can be used for porting Wasm programs to platforms for which no direct compiler exists.

We first consider the expected mapping from Wasm 3.0 to C/C++ below, mapping Wasm 3.0 unordered accesses to RC11 relaxed atomic accesses, Wasm release/acquire accesses to RC11 release/acquire accesses (release stores, acquire loads, acquire-release RMWs), and Wasm 3.0 sc accesses/fences to RC11 sc accesses/fences.

$$\begin{array}{lll} (\!|\mathsf{R}^{\mathsf{un}}(x,v)|\!)_{\mathsf{W}\to\mathsf{C}} \triangleq \mathsf{R}^{\mathsf{rlx}}(x,v) & (\!|\mathsf{R}^{\mathsf{ra}}(x,v)|\!)_{\mathsf{W}\to\mathsf{C}} \triangleq \mathsf{R}^{\mathsf{acq}}(x,v) & (\!|\mathsf{R}^{\mathsf{sc}}(x,v)|\!)_{\mathsf{W}\to\mathsf{C}} \triangleq \mathsf{R}^{\mathsf{sc}}(x,v) \\ (\!|\mathsf{W}^{\mathsf{un}}(x,v)|\!)_{\mathsf{W}\to\mathsf{C}} \triangleq \mathsf{W}^{\mathsf{rlx}}(x,v) & (\!|\mathsf{W}^{\mathsf{ra}}(x,v)|\!)_{\mathsf{W}\to\mathsf{C}} \triangleq \mathsf{W}^{\mathsf{rel}}(x,v) & (\!|\mathsf{W}^{\mathsf{sc}}(x,v)|\!)_{\mathsf{W}\to\mathsf{C}} \triangleq \mathsf{W}^{\mathsf{sc}}(x,v) \\ (\!|\mathsf{U}^{\mathsf{ra}}(x,v,v')|\!)_{\mathsf{W}\to\mathsf{C}} \triangleq \mathsf{U}^{\mathsf{acqrel}}(x,v,v') & (\!|\mathsf{U}^{\mathsf{sc}}(x,v,v')|\!)_{\mathsf{W}\to\mathsf{C}} \triangleq \mathsf{U}^{\mathsf{sc}}(x,v,v') & \\ (\!|\mathsf{F}|\!)_{\mathsf{W}\to\mathsf{C}} \triangleq \mathsf{F}^{\mathsf{sc}} & (\!|\mathsf{A}(x)|\!)_{\mathsf{W}\to\mathsf{C}} \triangleq \mathsf{A}(x) & (\!|\mathsf{P}(x)|\!)_{\mathsf{W}\to\mathsf{C}} \triangleq \mathsf{P}(x) \end{array}$$

This mapping is *mostly* correct in that it establishes all Wasm 3.0 axioms except for initialisation safety. The reason is that RC11 has no analogue of initialisation safety, and thus the only way to enforce initialisation safety is to map all accesses to release/acquire or stronger ones.

Theorem 6 (W→C). *The compilation scheme* $(\!|.|\!)_{\mathsf{W}\to\mathsf{C}}$ *from* Wasm 3.W *to* RC11 *is correct, where* Wasm 3.W *consistency is defined exactly as* Wasm 3.0 *consistency without the* INIT-SAFETY *axiom.*

Note that each Wasm 3.W event (e.g. a Wasm 3.W write) is mapped to the corresponding RC11 event (e.g. an RC11 write) with its counterpart mode (e.g. Wasm 3.W sc to RC11 sc, Wasm 3.W un

to RC11 rlx, and so forth). Moreover, the definition of Wasm 3.W consistency directly corresponds to that of RC11, provided that the Wasm 3.W access modes are replaced with corresponding RC11 ones, as described here. Therefore, the correctness of Theorem 6 is straightforward.

We next consider the stronger compilation scheme that maps unordered Wasm accesses to RC11 release/acquire accesses and other accesses as before:

$$(\!|\mathsf{R}^{\mathsf{un}}(x,v)|\!)_{\mathsf{W}\to\mathsf{C}'} \triangleq \mathsf{R}^{\mathsf{acq}}(x,v) \qquad (\!|\mathsf{R}^{\mathsf{ra}}(x,v)|\!)_{\mathsf{W}\to\mathsf{C}'} \triangleq \mathsf{R}^{\mathsf{acq}}(x,v) \qquad (\!|\mathsf{R}^{\mathsf{sc}}(x,v)|\!)_{\mathsf{W}\to\mathsf{C}'} \triangleq \mathsf{R}^{\mathsf{sc}}(x,v)$$

$$(\!|\mathsf{W}^{\mathsf{un}}(x,v)|\!)_{\mathsf{W}\to\mathsf{C}'} \triangleq \mathsf{W}^{\mathsf{rel}}(x,v) \qquad (\!|\mathsf{W}^{\mathsf{ra}}(x,v)|\!)_{\mathsf{W}\to\mathsf{C}'} \triangleq \mathsf{W}^{\mathsf{rel}}(x,v) \qquad (\!|\mathsf{W}^{\mathsf{sc}}(x,v)|\!)_{\mathsf{W}\to\mathsf{C}'} \triangleq \mathsf{W}^{\mathsf{sc}}(x,v)$$

$$(\!|\mathsf{U}^{\mathsf{ra}}(x,v,v')|\!)_{\mathsf{W}\to\mathsf{C}'} \triangleq \mathsf{U}^{\mathsf{acqrel}}(x,v,v') \qquad (\!|\mathsf{U}^{\mathsf{sc}}(x,v,v')|\!)_{\mathsf{W}\to\mathsf{C}'} \triangleq \mathsf{U}^{\mathsf{sc}}(x,v,v')$$

$$(\!|\mathsf{F}|\!)_{\mathsf{W}\to\mathsf{C}'} \triangleq \mathsf{F}^{\mathsf{sc}} \qquad (\!|\mathsf{A}(x)|\!)_{\mathsf{W}\to\mathsf{C}'} \triangleq \mathsf{A}(x) \qquad (\!|\mathsf{P}(x)|\!)_{\mathsf{W}\to\mathsf{C}'} \triangleq \mathsf{P}(x)$$

With this stronger compilation scheme, execution well-formedness implies that the initialisation writes of a dynamically allocated object x happen before every read from x. As per the coherence axiom of RC11, these reads then cannot read before an initialisation write.

Theorem 7 (W→C′)**.** *The compilation scheme* $(\!|.|\!)_{\mathsf{W}\to\mathsf{C}'}$ *from* Wasm 3.0 *to* RC11 *is correct.*

Note that the mapping in Theorem 7 is almost identical to that in Theorem 6, except that the Wasm 3.0 un accesses are compiled to stronger rel/acq accesses in RC11. As such, proving this compilation scheme sound is rather straightforward and follows closely from that of Theorem 6.

6 Correctness of Local Compiler Transformations

In what follows we prove that common local program transformations performed by compilers (comprising instruction reordering and elimination) are sound for *both* Wasm 3.0 *and* strong Wasm 3.0 consistency models. We do this for a standard set of transformations that have been studied by an extensive body of existing work in the literature [14, 33, 42, 66, 76]. As is standard, we establish our results at the *semantic* level, applying the transformations to the events in a given execution. That is, to demonstrate that a given transformation $P_{\text{src}} \rightsquigarrow P_{\text{trg}}$ is sound, we take an arbitrary Wasm 3.0-consistent (resp. strongly Wasm 3.0-consistent) execution G_{trg} of P_{trg} as a given, and prove that we can construct a Wasm 3.0-consistent (resp. strongly Wasm 3.0-consistent) execution G_{src} of P_{src} such that $\mathsf{Out}(G_{\text{trg}}) = \mathsf{Out}(G_{\text{src}})$. In doing so, we thus prove that every outcome of P_{trg} under (strong) Wasm 3.0 consistency is also an outcome of P_{src} under (strong) Wasm 3.0 consistency.

In addition to the conventional local transformations allowed by the C11 memory model [42, §7], we also prove the soundness of a more general form of *elimination of repeated load accesses* when both loads are unordered (un). This relies on the additional weakness of Wasm's unordered memory accesses over C/C++ 'relaxed' memory accesses. This then allows us to establish the soundness of the 'common sub-expression elimination' transformation (SubExp-Elim).

Strengthening. This transformation strengthens the memory order m of an event in the source to a stronger m′ ($\mathsf{m} \sqsubseteq \mathsf{m}'$) in the target. Proving this transformation sound is straightforward as (strong) Wasm 3.0 consistency is *monotone* with respect to $\sqsubseteq$ (see the extended version [61, Theorem 14 in §F]).

Sequentialisation. This transformation 'sequentialises' the program by merging two threads in the source G_{src} into one in the target G_{trg} by interleaving their events. Intuitively, this transformation adds edges to $G_{\text{trg}}.\mathsf{po}$; as such, its soundness follows straightforwardly from the monotonicity of (strong) Wasm 3.0 consistency with respect to po (see the extended version [61, Theorem 15 in §F]).

Deordering. We prove the soundness of the *deordering* transformation $\mathsf{X};\mathsf{Y} \rightsquigarrow \mathsf{X} \,||\, \mathsf{Y}$ for (strong) Wasm 3.0 consistency when X, Y constitute a *deorderable* pair (see the extended version [61,

X \ Y	$R_y^{m_2}$	$W_y^{m_2}$
$R_x^{m_1}$	$m_1 = un$	$m_1 = m_2 = un$
$W_x^{m_1}$	$m_1 \neq sc \vee m_2 \neq sc$	$m_2 = un$

$R^m; R^m \rightsquigarrow R^m$ $\quad$ $W^{sc}; R^{sc} \rightsquigarrow W^{sc}$ $\quad$ $U^m; R^{m'} \rightsquigarrow U^m$ $\quad$ $W^{m'}; U^m \rightsquigarrow W^{m'}$

$W^m; W^m \rightsquigarrow W^m$ $\quad$ $W^m; R^{ra} \rightsquigarrow W^m$ $\quad$ $U^m; U^m \rightsquigarrow U^m$ $\quad$ $F; F \rightsquigarrow F$

Fig. 2. Deorderable pairs of accesses X and Y, where x and y are distinct locations (above); mergeable pairs on accesses of the *same* location, where m' denotes the maximal mode in $\{ra, sc\}$ satisfying $m' \sqsubseteq m$ (below).

Theorems 16 and 17 in §F]). We define the conditions under which X, Y are deemed a deorderable pair in Fig. 2 (above). Note that X and Y are read or write accesses on *different* locations, and that RMW (update) accesses cannot be deordered. Observe that the *reordering* transformation $X; Y \rightsquigarrow Y; X$ can be obtained by applying deordering followed by sequentialisation: $X; Y \rightsquigarrow X \,||\, Y \rightsquigarrow Y; X$.

Merging. We prove that the *merging* transformation $X; Y \rightsquigarrow X$ eliminating Y (where X, Y are either memory accesses on the same location or are both fences) is sound under (strong) Wasm 3.0 consistency when X, Y constitute a *mergeable* pair (see the extended version [61, Theorem 18 in §F]). We define the conditions under which X, Y is deemed a mergeable pair in Fig. 2 (below). Note that thanks to the strengthening transformation, the explicit modes referred to in Fig. 2 are *upper bounds*; e.g. $W_x^{ra}; R_x^{ra}$ can first be strengthened to $W_x^{sc}; R_x^{sc}$ and then merged to W_x^{sc}.

Register Promotion. We prove that the global 'register promotion' transformation is sound for (strong) Wasm 3.0 consistency (see the extended version [61, Theorem 19 in §F]), whereby all accesses to a given (shared) memory location x are replaced by those to a (thread-local) register, provided that all accesses to x are by the same thread. When proving this transformation sound at the level of an execution G, this amounts to removing all accesses to x from G, provided that they are all $G.\mathrm{po}$-related.

Unordered, Same-Location, Read-Read Deordering. Finally, we prove that deordering two unordered (un) reads on the same location is sound under Wasm 3.0 consistency (see the extended version [61, Theorem 20 in §F]). This then renders the common sub-expression elimination sound under Wasm 3.0 consistency.

7 The Data Race Freedom Guarantee

In order to demonstrate the utility of our Wasm 3.0 model, we should *ideally* show that our semantics of sc accesses is not overly weak. One common way to do this is by establishing a *data-race-freedom* (DRF) guarantee, ensuring that for 'good' programs that adhere to certain conditions, the weak behaviours allowed by Wasm 3.0 are not observable and the outcomes of the program are those under the strong and intuitive SC model [43].

A program is considered 'good' if it does not contain *data races* under SC. A program contains an SC data race if it has an execution G that contains an SC data race $\langle a, b\rangle$ with $a, b \in G.\mathsf{E}^{sc}$. A pair $\langle a, b\rangle$ in G constitutes an SC data race if a and b are *conflicting* accesses and $\langle a, b\rangle \notin \mathsf{hb}_{sc} \cup \mathsf{hb}_{sc}^{-1}$ with $\mathsf{hb}_{sc} \triangleq (G.\mathrm{po} \cup G.\mathrm{rf}^{sc})^+$. Events a, b are conflicting if they are distinct ($a \neq b$), access the same location ($\mathtt{loc}(a) = \mathtt{loc}(b)$) and at least one is a write or update event ($\{\mathtt{typ}(a), \mathtt{typ}(b)\} \cap \{\mathtt{W}, \mathtt{U}\} \neq \emptyset$).

The DRF theorem *would* then ensure that for all Wasm 3.0 programs P, if all data races of P under the SC model are confined to sc accesses, then its semantics under Wasm 3.0 coincides with that under SC. However, as the Wasm 3.0 model allows unrestricted load buffering behaviours

(LB+data and LB+ctrls), the DRF guarantee does not directly hold, as discussed in §2. Nevertheless, we can show that strengthening the model to strong Wasm 3.0 consistency (Def. 9) to forbid $\mathsf{po} \cup \mathsf{rf}$ cycles (and thus all instances of load-buffering) allows us to establish the desired DRF theorem (Theorem 8) with its full proof given in the extended version [61, Theorem 21 in §G].

Definition 17 (Conflicts and races). Given an execution G, two accesses $a, b \in G.\mathsf{E}$ are *conflicting* if $a \neq b$, $\mathtt{loc}(a) = \mathtt{loc}(b)$ and $\{\mathtt{typ}(a), \mathtt{typ}(b)\} \cap \{\mathtt{W}, \mathtt{U}\} \neq \emptyset$. A pair $\langle a, b\rangle$ constitutes a *race* under SC, written $\langle a, b\rangle \in \mathsf{race}$, if a and b are conflicting and $\langle a, b\rangle \notin \mathsf{hb}_{\mathsf{sc}} \cup \mathsf{hb}_{\mathsf{sc}}^{-1}$ with $\mathsf{hb}_{\mathsf{sc}} \triangleq (G.\mathsf{po} \cup G.\mathsf{rf}^{\mathsf{sc}})^+$.

A Wasm 3.0 execution G is *race-free* if: $\forall a, b.\ \langle a, b\rangle \in G.\mathsf{race} \Rightarrow \mathtt{ord}(a) = \mathtt{ord}(b) = \mathsf{sc}$.

Theorem 8 (Data race freedom (DRF)). *A* Wasm 3.0 *execution* $G = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{data}, \mathsf{addr}\rangle$ *is* SC*-consistent if* $\mathsf{acyc}(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})$ *holds. Given a* Wasm *program* P*:*

if all SC*-consistent executions of* P *are race-free*
then for all executions G *of* P*, if* G *is strongly* Wasm 3.0*-consistent, then* G *is also* SC*-consistent.*

8 Model Checking

We present Waver, our prototype model-checking implementation for concurrent Wasm 3.0 programs. In what follows, we discuss the algorithmic and engineering challenges of implementing Waver and report our results when running Waver on synthetic and realistic Wasm programs. Our prototype Waver implementation is available through our project page [62].

Stateless Model Checking. Waver is a *stateless model checker* [20]: it verifies a concurrent program by exploring all of its executions, without storing the ones already explored. Specifically, Waver verifies a program P by enumerating its Wasm 3.0-consistent executions, one at a time and incrementally, following the TruSt algorithm [36],which we describe below through an example.

Consider the RW+R+W program below:

$$\begin{array}{l} a :=_{\mathsf{sc}} x \\ y :=_{\mathsf{sc}} 1 \end{array} \;\Big\|\; b :=_{\mathsf{sc}} y \;\Big\|\; x :=_{\mathsf{sc}} 1 \qquad \text{(RW+R+W)}$$

TruSt verifies this program by enumerating its execution graphs in a depth-first manner. Starting from the empty graph, it executes the program (e.g. from left to right), and records alternative exploration options (i.e. rf and mo choices) along the way, as shown in Fig. 3.

When TruSt executes $a := x$, the read event $\mathtt{R}(x)$ is added to the graph (graph A). This event can only read 0, as only the initialiser write is present in the graph. Similarly, when TruSt executes $y := 1$, $\mathtt{W}(y, 1)$ is placed mo-after `init`, and no alternative mo-options are recorded (graph B).

When $\mathtt{R}(y)$ is next added, TruSt must explore two cases: one where $\mathtt{R}(y)$ reads from $\mathtt{W}(y, 1)$, and one where it reads from `init`, as both rf options lead to consistent graphs. As in standard depth-first search, TruSt continues with one of these options and explores the other later (by backtracking). Assuming it continues with graph C, it then adds $\mathtt{W}(x, 1)$ and obtains the full execution 1.

Observe, however, that $\mathtt{W}(x, 1)$ is also a possible rf source for $\mathtt{R}(x)$, but this rf choice could not have been discovered when $\mathtt{R}(x)$ was added to the graph. To address this, whenever a write is added, TruSt also examines whether any pre-existing graph reads can be revisited to read from this write. In the example above, $\mathtt{W}(x, 1)$ revisits $\mathtt{R}(x)$ leading to execution E. Note that events added after $\mathtt{R}(x)$ and not $(\mathsf{po} \cup \mathsf{rf})^+$-before $\mathtt{W}(x, 1)$ are removed, as these might depend on the value read by the revisited read (as is the case here). Graph E will lead to two more full executions (one for each value read by $\mathtt{R}(y)$), but we omit the exploration for brevity.

Returning to graph D, the exploration is analogous to that of C, with one key difference. Although TruSt will again examine revisiting $\mathtt{R}(x)$ from $\mathtt{W}(x, 1)$ in exploration D, revisiting again would also lead to graph E and therefore to duplication. To avoid this, whenever there are multiple

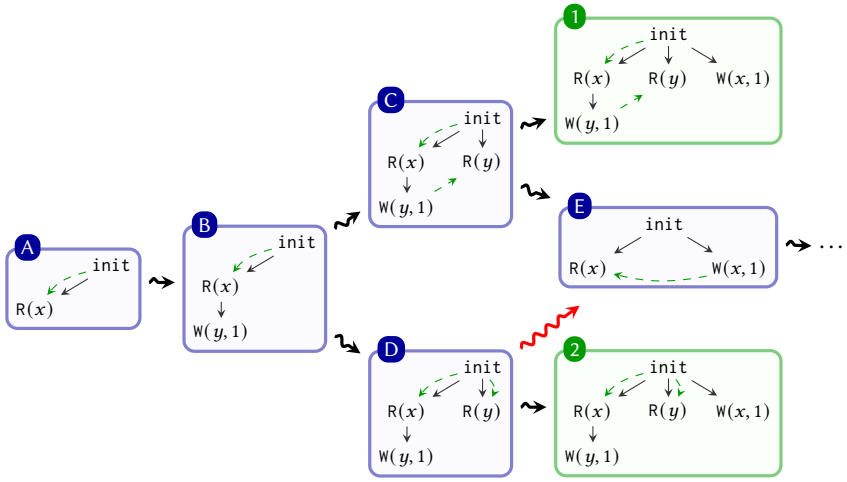


Fig. 3. An example run of TruSt, enumerating the executions of the RW+R+W program.

graphs in which the same revisit can occur, TruSt performs the revisit in only one of them (above, in graph C). As we briefly discuss below, TruSt employs a criterion that ensures revisit *uniqueness* and *existence*; the latter is crucial as different sub-explorations proceed completely independently.

Algorithmic Challenges: Extensibility. Although TruSt is parametric in the choice of the memory model, it cannot be blindly applied to Wasm 3.0. This is because TruSt requires the underlying model to be *maximally extensible*. Intuitively, extending a consistent graph G with a $(\mathsf{po} \cup \mathsf{rf})$-maximal read (resp. write) e leads to a consistent graph if e reads from (resp. is mo-after) the mo-maximal event in G. This property enables TruSt to construct graphs incrementally, but is also crucial in ensuring optimality: TruSt ensures revisit uniqueness and existence by requiring that all events deleted by a revisit be added maximally. As an example, recall the exploration of RW+R+W: revisiting $\mathsf{R}(x)$ takes place only from execution C, where $\mathsf{R}(y)$ reads from the mo-maximal write. In turn, reaching execution C from A is also guaranteed by extensibility.

While maximal extensibility is straightforward to prove for most consistency models, it does not immediately hold for Wasm 3.0 due to un (unordered) accesses not participating in mo. Concretely, a read r might not be able to consistently read from the mo-maximal write w, as w might have been overwritten by an unordered write w' that is hb-before r. Note that r cannot always consistently read from the write that was added last either: the last-added write w can be a $\sqsupseteq$ra write that is mo before another write w', which in turn is hb-before r.

Fortunately, however, we can show that un accesses preserve consistency, provided they are added with the recipe below. This recipe guarantees both revisit existence and uniqueness, as it incorporates a deterministic tie-braking criterion (addition order, denoted by $<$) to determine read maximality (there might be multiple un writes from which a read can consistently read).

Assume that executions are extended with a $<$ component to track the event addition order. For each location x, we define the visibility order $S \triangleq \mathsf{mo}_x \cup [\mathsf{W}_x]; \mathsf{hb}; [\mathsf{W}_x]; \mathsf{mo}_x^?$. Given an execution G and $e \in G.\mathsf{E}$, we write $\mathsf{SetMaxRF}(G, e)$ to denote an execution G' that differs from G only in that e reads from the $<$-latest S-maximal write to $\mathtt{loc}(e)$. Similarly, we write $\mathsf{SetMaxMO}(G, e)$ for a graph G' that only differs from G in that e is added mo-maximally. We write $G \setminus \{e\}$ to denote the execution obtained from G by removing e from G. We write $\mathsf{consistent}_{\mathsf{Wasm\ 3.0}}(G)$ to denote that G is Wasm 3.0-consistent.

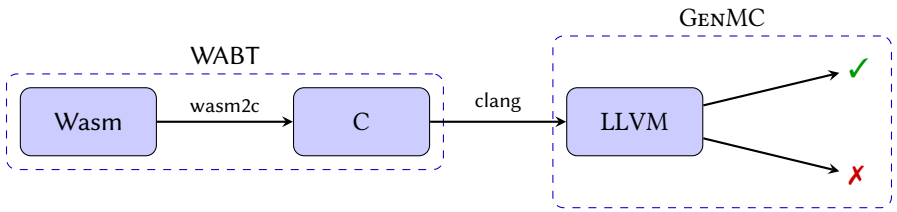


Fig. 4. The architecture of Waver

Theorem 9. Wasm 3.0 *is* maximally extensible*: for all executions* G *and* $e \in G.\mathsf{E}$ *such that* e *is* $(\mathsf{po} \cup \mathsf{rf})^+$*-maximal and* $\mathsf{consistent}_{\text{Wasm 3.0}}(G \backslash \{e\})$*, the following hold:*

- *if* $e \in G.\mathsf{R} \setminus \mathsf{U}$*, then* $\mathsf{consistent}_{\text{Wasm 3.0}}(\mathsf{SetMaxRF}(G, e))$
- *if* $e \in G.\mathsf{W} \setminus (\mathsf{W}^{\mathsf{un}} \cup \mathsf{U})$*, then* $\mathsf{consistent}_{\text{Wasm 3.0}}(\mathsf{SetMaxMO}(G, e))$
- *if* $e \in G.\mathsf{U}$*, then* $\mathsf{consistent}_{\text{Wasm 3.0}}(\mathsf{SetMaxRF}(G, \mathsf{SetMaxMO}(G, e)))$
- *if* $e \notin G.\mathsf{R} \cup G.\mathsf{W}$ *or* $e \in G.\mathsf{W}^{\mathsf{un}}$*, then* $\mathsf{consistent}_{\text{Wasm 3.0}}(G)$

Proof (sketch). It is easy to see that S is acyclic in any consistent execution (mo totality), and there is a finite set of S-maximal elements for each location x. Given a location x, we call the $<$-largest S-maximal element w_{max_x}. Any consistency violation has to be due to the new event e. Below, we focus on the read and unordered-write cases; the others trivially follow.

Case $e \in G.\mathsf{R} \setminus \mathsf{U}$**:** init-safety holds for G' due to well-formedness of G. coherence and sc follow due to the S-maximality of w_{max_x}: the only outgoing edges from e are rb edges, but $[e]; G'.\mathsf{rb} = \emptyset$, as rb-successors of e are exactly the S-successors of w_{max_x}, i.e. the empty set.

Case $e \in G.\mathsf{W}^{\mathsf{un}}$: The only non-trivial case is due to rb edges into e due to $\mathsf{hb}_{\mathsf{loc}}$. A coherence violation requires outgoing $\mathsf{hb}_{\mathsf{loc}}$ edges from e, which is impossible; an sc violation requires $e \in G.\mathsf{W}^{\mathsf{sc}}$, which contradicts our assumptions.

□

Engineering Challenges. We implemented Waver on top of the GenMC framework [38], a state-of-the-art implementation of TruSt that accepts LLVM bitcode programs as input. GenMC executes programs using the interpreter lli, and can accommodate the addition of new models.

Our first challenge was the absence of an interpreter that can execute concurrent Wasm programs. As such, we decided to use GenMC's existing lli interpreter and convert Wasm programs to LLVM bitcode in a semantics-preserving way using WABT [22]. WABT is a suite of Wasm utilities that includes wasm2c, a tool that converts Wasm to C code, which is then passed to GenMC as input.

A second challenge was that wasm2c supports neither Wasm 3.0 nor the concurrency features we discussed earlier. To run Waver on programs similar to those in §2, we extended WABT with support for GC opcodes, so that Wasm 3.0 programs parse and type-check, and wasm2c generates appropriate C code. The resulting architecture is depicted in Fig. 4.

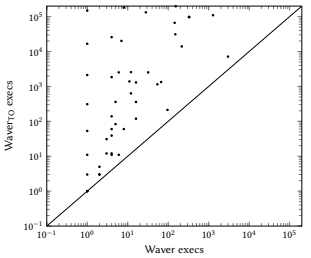
A final challenge was that, although GenMC is consistency-model-agnostic in theory, its implementation assumes certain properties from the model in order to make verification more effective. One such property is coherence, a.k.a. SC-per-location. Intuitively, most consistency violations violate coherence as opposed to other axioms: e.g. if a thread reads some value, it will then only be able to read the same or more recent values. As such, GenMC first checks whether a candidate rf/mo option for a new event is coherent (such checks are cheap) before checking full consistency.

For Wasm 3.0, however, since unordered accesses render model non-coherent, we modified GenMC's implementation so that this filter properly handles unordered accesses, and accordingly adjusted its extensibility definitions.

Evaluation. As Waver is the *first* model checker for concurrent Wasm, we could not evaluate it against other tools. Moreover, there is currently *no* Wasm 3.0 *code* that exercises its concurrency features as these are still under development. As such, we evaluated Waver in three ways.

First, to gain confidence in our translation infrastructure, we manually wrote a number of litmus tests (similar to those in §2) directly in Wasm 3.0, and Waver enumerated their executions almost *instantly*. We also implemented a few simple spinlock implementations in Wasm 3.0 and considered their client programs comprising up to eight threads. Waver exhaustively explored the state space of these clients (an order of $N!$ executions, where N is the number of threads) in under an hour.

Second, to demonstrate the impact of our mo-based Wasm 3.0 model in model checking, we compare Waver against Waver_{TO}, a version of Waver that tracks a total order tot to check Wasm 3.0-consistency. For that, we used a collection of 50 synthetic tests. The scatter plot on the right demonstrates that Waver explores exponentially fewer executions than Waver_{TO} (highlighting the efficiency of our model formulation), while Waver_{TO} fails to scale for programs longer than a few lines of code.



Third, as implementing larger pieces of concurrent software directly in Wasm 3.0 is arduous, we ran our modified GenMC implementation on concurrent data structures written in C, but using the Wasm 3.0 model. Overall, GenMC's performance under Wasm 3.0 was no different than its performance under existing C/C++ models such as RC11. We have passed Waver to industrial engineers involved in the implementation of the *shared-everything threads* project [73], and we are continuing to engage with them as they experiment with Waver internally.

9 Discussion, Related Work and Conclusions

We have presented a formal account of the concurrency semantics of Wasm 3.0 and have demonstrated that it meets all desired and relevant formal results. We have carried out our work in close collaboration with members of *WebAssembly's industrial standards community*, with the aim that our semantics should be adopted as the official concurrency model for Wasm 3.0 as the *shared-everything threads* project progresses. Notably, prototypes based on our Wasm 3.0 model are under active implementation in a Wasm compiler by a major industrial stakeholder [73].

Due to the specific design constraints of Wasm, our model is rather different from existing weakly consistent models for other programming languages [9, 18, 42, 49, 80, 81]. Nevertheless, it is based on the Wasm 1.0 model and has drawn inspiration from features in the C/C++ and Java models, which we discuss shortly.

By occasionally requiring $\mathsf{po} \cup \mathsf{rf}$ acyclicity, our model has only superficially addressed the main open problem of preventing 'out of thin air' (OOTA) behaviours without inducing a performance penalty or additional obstacles in reasoning about program correctness. While there exist several more advanced proposals for resolving the OOTA problem [14, 28–30, 33, 45, 52, 56], none satisfy all the desiderata for widespread adoption. Given the state of the art in resolving the OOTA problem, we believe that requiring $\mathsf{po} \cup \mathsf{rf}$ acyclicity throughout, despite its non-zero cost, is a good compromise.

C/C++ *Concurrency.* C and C++ first introduced an almost identical formal concurrency model in their 2011 standards, even though both were widely used for concurrent programming much before then. Batty et al. [9] first formalised this model; others [42, 48, 76] subsequently noted several errors in the original model and proposed corrections, which were largely incorporated in later C++ standards. Unfortunately, one correction concerning the interaction between sc and release/acquire (ra) atomics was addressed incorrectly in the C++20 standard, resulting in an overly strong model.

This is still in the process of being rectified to agree with academic research. Our treatment of sc and ra accesses follows closely that of the *repaired C11* (RC11) model [42]. This is necessary because we want the RC11 and Wasm models to agree when restricted to sc and ra accesses, so as to allow the identity mapping between these models to be correct (Theorems 4 and 7).

Java Concurrency. Java embraced weak memory consistency much before C/C++ with Manson et al. [49] defining the very influential Java memory model (JMM) formally back in 2005. Very much like the Wasm 1.0 model, this initial Java model only supported unordered accesses (without read-read coherence) and sc accesses. Unlike Wasm 1.0, however, the Java model was defined in a much more complex way in an attempt to avoid OOTA behaviours. Sadly, the model was shown to be flawed [67] in several ways that could not be repaired with local changes. Java's model was later updated to adopt C/C++'s different memory orders, but left open the issue of OOTA outcomes [44].

Our initialisation safety guarantee is inspired by Java's purported semantics for 'final' fields, which are supposed to be initialised by the object constructor and cannot be updated later. As long as no reference to an object is escaped during its construction, no thread can see an object with an uninitialised final field. To our knowledge, despite its impressive mechanisation [46], there are no formal compilation results from the Java memory model to weak architectures such as Armv8.

JavaScript and Wasm *Concurrency.* The concurrency models of Wasm [81] and JavaScript (JS) [80] are inextricably linked. On the web, the shared buffers that underpin Wasm 1.0 concurrency are also exposed as *SharedArrayBuffer* objects in JS—these are the only mechanism for shared memory concurrency in JS [79]. The two languages' models must therefore be fully interoperable, with the same observable semantics [80]. Similarly, the industrial effort to extend Wasm 3.0 with concurrency is mirrored by an early-stage proposal to introduce shareable, typed structs to JS [84]. It is likely that the concurrency model adopted for Wasm 3.0 will be mirrored by JS.

Both Wasm (as discussed here) and JS use C-style per-candidate-execution axiomatic models and are thus vulnerable to OOTA issues in the absence of an axiom such as $\mathsf{po} \cup \mathsf{rf}$ acyclicity.

Mattarei et al. [51] present a model checker for an early version of JS based on Alloy [27], capable of generating executions for small program fragments containing around eight memory events. Watt et al. [80] present another Alloy-based model for a revised version of JS that can similarly only handle small program fragments. They also prove correct compilation to IMM [57] for the fragment of the JS model equivalent to the Wasm 1.0 model restricted to non-mixed-size executions.

Model Checking for Weak Memory Concurrency. There has been a long sequence of results on model checking for weak memory models. Early works targetted a specific consistency model, such as x86-TSO [1] or RC11 [34, 55]. Kokologiannakis et al. [36, 37] later introduced stateless model checking algorithms that were parametric in the choice of an axiomatic memory model, as long as it satisfies prefix-closedness and (maximal) extensibility, and can therefore be applied to our Wasm 3.0 model fairly easily. An alternative approach, which could also be applied to Wasm 3.0, is SMT-based bounded model checking such as `dartagnan` [19].

Other Work on Weak Memory Consistency. The literature also contains a number of formal memory model definitions for (specific features of) hardware architectures, programming languages or intermediate languages, e.g. Sparc TSO/PSO/RMO [69], x86-TSO [68], Arm [58], Power [47, 65], RISC-V [59], CXL [71], FPGAs [26], IMM [57], Inline-Asm [16], LKMM [4], LLVM [13], NT stores [63], NVIDIA GPUs [2], OCaml [18], OpenCL [8], SRA [39], RDMA [6], Rust [15]. While most such definitions are in a per-execution axiomatic style, some definitions are presented in an operational fashion using special constructs for modelling instruction reordering and/or propagation delays (e.g. buffers, message timestamps, thread and message views, promises, re-execution, event structures). A few other works have explored more exotic definition styles, such as denotational [30] and

transformation-based definitions [41]. Overall, axiomatic definitions are clearly the most widely used ones because they offer important benefits over other definition styles. They are typically more concise, easier to compare with one another even in an automated fashion [35, 83], easier to reason about (e.g. to show compilation and transformation correctness) and more suitable for automated exploration techniques [5, 37]. For this reason, very often, when a paper uses a different definition style (e.g. operational), it proves that that definition is equivalent to an axiomatic one.

Similarly, there is substantial work on reasoning about programs in a weak memory setting, both manually and automatically. Doko and Vafeiadis [17], Kaiser et al. [31], Lahav and Vafeiadis [40], Turon et al. [75], Vafeiadis and Narayan [77] have developed program logics for reasoning about RC11 programs, while Alglave and Cousot [3], Hammond et al. [25], Svendsen et al. [70] have proposed similar logics for weaker models. Other works have considered the decidability/complexity of various verification problems such as consistency checking [12, 74], error reachability [7] and robustness [10], as well as more practical approximations for robustness [11, 21, 50].

Data Availability Statement. As stated in §8, Waver has been implemented on top of the open-source GenMC tool. It will be made publicly available as an open-source archive [62].

Acknowledgments

Azalea Raad is supported by the UKRI Future Leaders Fellowship MR/V024299/1, the EPSRC grant EP/X037029/1 and VeTSS. Conrad Watt is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 1 (RG18/25). We thank Thomas Lively and the OOPSLA 2026 reviewers for their feedback and suggestions, which have greatly improved the quality of this paper.

References

[1] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless model checking for TSO and PSO. In *TACAS 2015 (LNCS, Vol. 9035)*. Springer, Berlin, Heidelberg, 353–367. https://doi.org/10.1007/978-3-662-46681-0_28

[2] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *ASPLOS 2015*. ACM, 577–591. https://doi.org/10.1145/2694344.2694391

[3] Jade Alglave and Patrick Cousot. 2017. Ogre and Pythia: an invariance proof method for weak consistency models. In *POPL 2017*. ACM, 3–18. https://doi.org/10.1145/3009837.3009883

[4] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *ASPLOS 2018*. ACM, 405–418. https://doi.org/10.1145/3173162.3177156

[5] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. https://doi.org/10.1145/2627752

[6] Guillaume Ambal, Brijesh Dongol, Haggai Eran, Vasileios Klimis, Ori Lahav, and Azalea Raad. 2024. Semantics of Remote Direct Memory Access: Operational and Declarative Models of RDMA on TSO Architectures. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1982–2009. https://doi.org/10.1145/3689781

[7] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 7–18. https://doi.org/10.1145/1706299.1706303

[8] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *POPL 2016*. ACM, 634–648. https://doi.org/10.1145/2837614.2837637

[9] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL 2011*. ACM, 55–66. https://doi.org/10.1145/1926385.1926394

[10] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. 2011. Deciding robustness against total store ordering. In *ICALP 2011 (LNCS, Vol. 6756)*. Springer, 428–440. https://doi.org/10.1007/978-3-642-22012-8_34

[11] Soham Chakraborty. 2021. Robustness between Weak Memory Models. In *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 173–182. https://doi.org/10.34727/2021/ISBN.978-3-85448-046-4_26

[12] Soham Chakraborty, Shankara Narayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. 2024. How Hard Is Weak-Memory Testing? *Proc. ACM Program. Lang.* 8, POPL (2024), 1978–2009. https://doi.org/10.1145/3632908

[13] Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the concurrency semantics of an LLVM fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. ACM, 100–110. http://dl.acm.org/citation.cfm?id=3049844

[14] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 70:1–70:28. https://doi.org/10.1145/3290383

[15] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

[16] Paulo Emílio de Vilhena, Ori Lahav, Viktor Vafeiadis, and Azalea Raad. 2024. Extending the C/C++ Memory Model with Inline Assembly. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1081–1107. https://doi.org/10.1145/3689749

[17] Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *VMCAI 2016 (LNCS, Vol. 9583)*. Springer, 413–430. https://doi.org/10.1007/978-3-662-49122-5_20

[18] Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding data races in space and time. In *PLDI 2018*. ACM, 242–255. https://doi.org/10.1145/3192366.3192421

[19] Natalia Gavrilenko, Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for weak memory models: Relation analysis for compact SMT encodings. In *CAV 2019*. Springer, Cham, 355–365. https://doi.org/10.1007/978-3-030-25540-4_19

[20] Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *POPL 1997* (Paris, France). ACM, New York, NY, USA, 174–186. https://doi.org/10.1145/263699.263717

[21] Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. 2022. Checking robustness to weak persistency models. In *PLDI 2022*. ACM, 490–505. https://doi.org/10.1145/3519939.3523723

[22] WebAssembly Community Group. 2025. WABT: The WebAssembly Binary Toolkit. https://github.com/WebAssembly/wabt/

[23] WebAssembly Community Group. 2025. wasm2c: Convert Wasm files to C source and header. https://github.com/WebAssembly/wabt/tree/main/wasm2c

[24] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *PLDI 2017*. ACM, 185–200. https://doi.org/10.1145/3062341.3062363

[25] Angus Hammond, Zongyuan Liu, Thibaut Pérami, Peter Sewell, Lars Birkedal, and Jean Pichon-Pharabod. 2024. An Axiomatic Basis for Computer Programming on the Relaxed Arm-A Architecture: The AxSL Logic. *Proc. ACM Program. Lang.* 8, POPL (2024), 604–637. https://doi.org/10.1145/3632863

[26] Dan Iorga, Alastair F. Donaldson, Tyler Sorensen, and John Wickerson. 2021. The semantics of shared memory in Intel CPU/FPGA systems. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. https://doi.org/10.1145/3485497

[27] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (April 2002), 256–290. https://doi.org/10.1145/505145.505149

[28] Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with preconditions: A simple model of relaxed memory. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 194:1–194:30. https://doi.org/10.1145/3428262

[29] Alan Jeffrey and James Riely. 2019. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. *Log. Methods Comput. Sci.* 15, 1, Article 33 (2019), 25 pages. https://doi.org/10.23638/LMCS-15(1:33)2019

[30] Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The leaky semicolon: Compositional semantic dependencies for relaxed-memory concurrency. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498716

[31] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP 2017 (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29. https://doi.org/10.4230/LIPICS.ECOOP.2017.17

[32] Sangeeta Kakati and Mats Brorsson. 2023. WebAssembly Beyond the Web: A Review for the Edge-Cloud Continuum. In *2023 3rd International Conference on Intelligent Technologies (CONIT)*. 1–8. https://doi.org/10.1109/CONIT59222.2023.10205816

[33] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL 2017*. ACM, 175–189. https://doi.org/10.1145/3009837.3009850

[34] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL (2018), 17:1–17:32. https://doi.org/10.1145/3158105

[35] Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. 2023. Kater: Automating Weak Memory Model Metatheory and Consistency Checking. *Proc. ACM Program. Lang.* 7, POPL (2023), 544–572. https://doi.org/10.1145/3571212

[36] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. 2022. Truly stateless, optimal dynamic partial order reduction. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498711

[37] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model checking for weakly consistent libraries. In *PLDI 2019*. ACM, 96–110. https://doi.org/10.1145/3314221.3314609

[38] Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A Model Checker for Weak Memory Models. In *CAV 2021 (LNCS, Vol. 12759)*. Springer, 427–440. https://doi.org/10.1007/978-3-030-81685-8_20

[39] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *POPL 2016*. ACM, 649–662. https://doi.org/10.1145/2837614.2837643

[40] Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *Automata, Languages, and Programming*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 311–323. https://doi.org/10.1007/978-3-662-47666-6_25

[41] Ori Lahav and Viktor Vafeiadis. 2016. Explaining Relaxed Memory Models with Program Transformations. In *FM 2016 (LNCS, Vol. 9995)*. Springer, 479–495. https://doi.org/10.1007/978-3-319-48989-6_29

[42] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI 2017*. ACM, 618–632. https://doi.org/10.1145/3062341.3062352

[43] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[44] Doug Lea. 2013. *JEP 188: Java Memory Model Update*. Java Enhancement Proposal 188. OpenJDK, OpenJDK. https://openjdk.org/jeps/188

[45] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: global optimizations in relaxed memory concurrency. In *PLDI 2020*. ACM, 362–376. https://doi.org/10.1145/3385412.3386010

[46] Andreas Lochbihler. 2012. Java and the Java Memory Model – a Unified, Machine-Checked Formalisation. In *ESOP 2012 (LNCS, Vol. 7211)*. Springer, 497–517. https://doi.org/10.1007/978-3-642-28869-2_25

[47] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *CAV 2012 (LNCS, Vol. 7358)*. Springer, 495–512. https://doi.org/10.1007/978-3-642-31424-7_36

[48] Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2016. Counterexamples and Proof Loophole for the C/C++ to POWER and ARMv7 Trailing-Sync Compiler Mappings. https://doi.org/10.48550/arXiv.1611.01507 arXiv:1611.01507 [cs.PL]

[49] Jeremy Manson, William W. Pugh, and Sarita V. Adve. 2005. The Java memory model. In *POPL 2005*. ACM, 378–391. https://doi.org/10.1145/1040305.1040336

[50] Roy Margalit and Ori Lahav. 2021. Verifying observational robustness against a C11-style memory model. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–33. https://doi.org/10.1145/3434285

[51] Cristian Mattarei, Clark Barrett, Shu-yu Guo, Bradley Nelson, and Ben Smith. 2018. EMME: A Formal Tool for ECMAScript Memory Model Evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 55–71. https://doi.org/10.1007/978-3-319-89963-3_4

[52] Evgenii Moiseenko, Matteo Meluzzi, Innokentii Meleshchenko, Ivan Kabashnyi, Anton Podkopaev, and Soham Chakraborty. 2025. Relaxed Memory Concurrency Re-executed. *Proc. ACM Program. Lang.* 9, POPL (2025), 2149–2175. https://doi.org/10.1145/3704908

[53] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting fine grain isolation in the firefox renderer. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 40, 18 pages.

[54] Olivier Nicole and Isabella Leandersson. 2025. The First Wasm_of_ocaml Release is Out! https://tarides.com/blog/2025-02-19-the-first-wasm-of-ocaml-release-is-out/.

[55] Brian Norris and Brian Demsky. 2016. A Practical Approach for Model Checking C/C++11 Code. *ACM Trans. Program. Lang. Syst.* 38, 3 (2016), 10:1–10:51. https://doi.org/10.1145/2806886

[56] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *ESOP 2020 (LNCS, Vol. 12075)*. Springer, 599–625. https://doi.org/10.1007/978-3-030-44914-8_22

[57] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* 3, POPL (2019), 69:1–69:31. https://doi.org/10.1145/3290382

[58] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

[59] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: A simpler and faster operational concurrency model. In *PLDI 2019* (Phoenix, AZ, USA). ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/3314221.3314624

[60] Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. 2019. On Library Correctness under Weak Memory Consistency: Specifying and Verifying Concurrent Libraries under Declarative Consistency Models. *Proc. ACM Program. Lang.* 3, POPL, Article 68 (jan 2019), 31 pages. https://doi.org/10.1145/3290381

[61] Azalea Raad, Michalis Kokologiannakis, Viktor Vafeiadis, and Conrad Watt. 2026. *A Formal Account of the Wasm 3.0 Concurrency Model (extended version).* Technical Report. https://www.soundandcomplete.org/papers/OOPSLA2026/Wasm/Wasm-extended.pdf

[62] Azalea Raad, Michalis Kokologiannakis, Viktor Vafeiadis, and Conrad Watt. 2026. Project Page: A Formal Account of the Wasm 3.0 Concurrency Model. https://www.soundandcomplete.org/papers/OOPSLA2026/Wasm

[63] Azalea Raad, Luc Maranget, and Viktor Vafeiadis. 2022. Extending Intel-x86 consistency and persistency: formalising the semantics of Intel-x86 memory types and non-temporal stores. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498683

[64] Andreas Rossberg. 2023. Mutually Iso-Recursive Subtyping. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 347–373. https://doi.org/10.1145/3622809

[65] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *PLDI 2011*. ACM, 175–186. https://doi.org/10.1145/1993498.1993520

[66] Jaroslav Ševčík. 2011. Safe optimisations for shared-memory concurrent programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 306–316. https://doi.org/10.1145/1993498.1993534

[67] Jaroslav Sevcík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP 2008 (LNCS, Vol. 5142)*. Springer, 27–51. https://doi.org/10.1007/978-3-540-70592-5_3

[68] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. https://doi.org/10.1145/1785414.1785443

[69] SPARC International Inc. 1994. *The SPARC architecture manual (version 9)*. Prentice-Hall.

[70] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A Separation Logic for a Promising Semantics. In *ESOP 2018 (LNCS, Vol. 10801)*. Springer, 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

[71] Chengsong Tan, Alastair F. Donaldson, and John Wickerson. 2025. Formalising CXL Cache Coherence. In *ASPLOS 2025*. ACM, 437–450. https://doi.org/10.1145/3676641.3715999

[72] Michael Thomas and Thomas Steiner. 2024. Why Google Sheets ported its calculation worker from JavaScript to WasmGC. https://web.dev/case-studies/google-sheets-wasmgc.

[73] Chromium Issue Tracker. 2024. WebAssembly Shared-Everything Threads proposal implementation tracking bug. https://issues.chromium.org/issues/42204563.

[74] Hünkar Can Tunç, Parosh Aziz Abdulla, Soham Chakraborty, Shankaranarayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. 2023. Optimal Reads-From Consistency Checking for C11-Style Memory Models. *Proc. ACM Program. Lang.* 7, PLDI (2023), 761–785. https://doi.org/10.1145/3591251

[75] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA 2014*. ACM, 691–707. https://doi.org/10.1145/2660193.2660243

[76] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *POPL 2015*. ACM, 209–220. https://doi.org/10.1145/2676726.2676995

[77] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA 2013*. ACM, 867–884. https://doi.org/10.1145/2509136.2509532

[78] Stefan Wallentowitz, Bastian Kersting, and Dan Mihai Dumitriu. 2022. Potential of WebAssembly for Embedded Systems. In *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. 1–4. https://doi.org/10.1109/MECO55406.2022.9797106

[79] Conrad Watt. 2025. Concurrency in WebAssembly: Experiments in the web and beyond. *Queue* 23, 3 (July 2025), 65–85. https://doi.org/10.1145/3747201.3746173

[80] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. 2020. Repairing and mechanising the JavaScript relaxed memory model. In *PLDI 2020*. ACM, 346–361. https://doi.org/10.1145/3385412.3385973

[81] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. 2019. Weakening WebAssembly. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 133:1–133:28. https://doi.org/10.1145/3360559

[82] WebAssembly Community Group. 2025. Shared-everything threads. https://github.com/WebAssembly/shared-everything-threads.

[83] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *POPL 2017*. ACM, 190–204. https://doi.org/10.1145/3009837.3009838

[84] Shu yu Guo. 2025. JavaScript Structs. https://github.com/tc39/proposal-structs.

Received 2026-03-22; accepted 2026-06-10