Burrow: A Proof Framework for Weak Memory

Dennis Sprokholt[1]* and Soham Chakraborty[2]

[1] TU Munich, Germany
dennis.sprokholt@tum.de
[2] TU Delft, Netherlands
s.s.chakraborty@tudelft.nl

Abstract. Burrow is a proof framework for weak memory mapping proofs. Those mappings appear as optimizations and translations between languages inside compilers and binary translators. However, their mechanized proofs, when defined over formal axiomatic weak memory semantics, are often large and complex. In this paper, we discuss the proof primitives provided by Burrow which simplify mechanizing those mapping proofs and help to prove many lemmas *generally*. To demonstrate the benefits of these primitives, we use Burrow to prove a mapping from x86 to Arm correct.

1 Introduction

When writing a concurrent program, programmers and compilers must consider how threads could interleave arbitrarily [21]. In addition to interleavings, hardware also shows *weak* behaviors [29,5], observable when instructions execute out-of-order or synchronization between CPU cores delays. As these weak memory behaviors differ between architectures and high-level programming languages [7,6,20], compilers must ensure not to introduce new weak behaviors that the original program could not show.

The only way to know whether a program translation or transformation is correct is to inspect the weak memory semantics [5,7] of the corresponding architectures and languages – these models are often large and complex. In addition, weak memory requires *global reasoning* about a program, as any thread could influence the order within which *any other* thread observes changes to memory. Through formal proofs we can reason about the correctness of these transformations. However, these proofs involve many cases with subtle complexities and are often error-prone [37,36,6,25].

Mechanizing proofs in a proof assistant increases the trustworthiness of correctness claims over pen-and-paper proofs, but writing those proofs is very time-consuming. Particularly, weak memory mapping proofs must guarantee the mapping preserves many properties, totaling around 30 axioms (which include well-formedness conditions). Proofs for the majority of those axioms are similar across mappings, resulting in uninteresting "boilerplate code"; refactoring

* Corresponding author. Work done while the author was at TU Delft.

proofs for their new context is often time-consuming [1,9,12]. However, fragments of each proof that are unique to the specific mapping must always be written from scratch (*i.e.* when proving consistency).

Burrow is a framework for axiomatic weak memory mapping proofs, implemented in the Agda proof assistant [2,28]. Burrow provides primitives and abstractions, addressing formalization patterns common when proving axiomatic weak memory transformations by contradiction. The abstractions greatly reduce the mechanization effort and "boilerplate code" needed for weak memory proofs. In addition, the primitives make it easier to prove those fragments unique to each mapping, for instance, by translating common predicates and relations along the mapping; from source-to-target and from target-to-source.

The theory of axiomatic weak memory is well established [7,5]. Instead, this paper covers our solutions to the weak memory *mechanization challenges*, both in general and specifically in Agda. The design goals of Burrow were:

- *General*, as it should be able to handle mapping proofs between axiomatic weak memory models for any pair of languages and/or architectures, while also supporting program transformation proofs.
- *Extensible*, to easily support proofs for weak memory features that extend the base models; for instance, mixed-size accesses [13,4] and scopes [23].
- *Comprehensive*, where it includes mapping primitives and lemmas for features common within mapping proofs.
- *Readable*, meaning the provided primitives and idiomatic proof style enforced by Burrow should be close to the formal models in mathematical notation.

We achieve these design goals in Burrow as follows:

- Burrow's *modular design* includes an opinionated framework that proves many axioms generally, greatly simplifying mapping proofs between languages. As optimization proofs often require optimization-specific lemmas, we also provide a more general framework that still proves various axioms generally (§ 3.2).
- Semantics for any language can be specified in their common structure (§ 3.1), obtained from theory [5,7], for which Burrow defines well-formedness axioms generally, while also providing lemmas for models and mappings (§ 3.2).
- Burrow provides proof primitives that conveniently translate properties and relations along the mapping, which happens countlessly in proofs (§ 3.3). In addition, our separate proof library Dodo contains lemmas for binary relations beyond weak memory.
- Burrow is implemented in Agda, improving proof readability. By writing proofs directly in the term language, those proof cases covered by each lemma are made very explicit in the syntax (by example throughout § 3).

We used early versions of Burrow to prove mapping schemes used in our binary translators Lasagne [34], Risotto [14], and Arancini [32]. Throughout those project we proved mappings from x86 [29] to Arm [31,5,4] and RISC-V [33]. Notably, through those efforts we found errors in QEMU, and also an issue with the Arm model [14, §3.3]. While mechanizing proofs with Burrow for Arancini [32] we also discovered that mixed-size accesses [13] cannot be split.

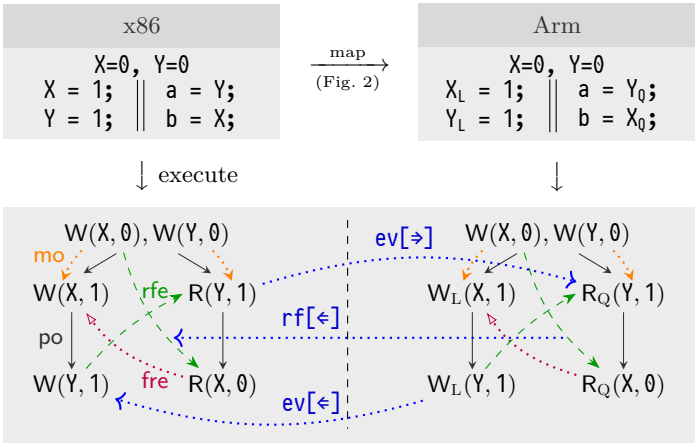

Fig. 1: Illustration of the *message passing* program mapped from x86 to Arm (top), with execution graphs generated by both (bottom). The po, rf, mo, and fr edges originate from the theory, while the blue arrows illustrate mapping primitives in Burrow (*e.g.*, ev[⇒]), which are used extensively in consistency proofs.

2 Background & Motivation

As Burrow is a framework for writing weak memory proofs in Agda, we first introduce both weak memory and Agda.

2.1 Weak Memory

Axiomatic weak memory models represent program semantics with a *set of execution graphs restricted by axioms* [5]. An execution consists of *events*, each representing either read accesses to memory (R), write accesses (W), or fences (F) that enforce an order between accesses. Events are connected by relations, for instance, *reads-from* (rf) relates a W event to each R event reading its value. We consider the conventional primitive relations: program order (po), reads-from (rf), modification order (mo), and read-modify-write (rmw). From-read (fr) is *derived* as $\mathsf{fr} \triangleq \mathsf{rf}^{-1};\mathsf{mo}$. Finally, when relations connect events in different threads, they are *external*; for instance, rfe, moe, and fre. Fig. 1 shows the *message passing* program on x86 and Arm, each with an execution graph containing po, rfe, mo, and fre edges.

Well-formedness. Execution graphs must be *well-formed*, which intuitively states they "make sense". For instance, every R event reads-from (rf) *exactly one* W event; and when one W overwrites another W—captured with mo—then both those events must surely access the same memory location.

Although well-formedness constraints are often implicitly assumed to hold when writing pen-and-paper proofs, with proof mechanization they must be defined explicitly. We identified 27 primitive well-formed properties, which we

mechanized in Burrow. Several dozen additional general well-formedness properties we *derive* from those primitive properties.

Consistency. Each architecture and language defines their own *consistency axioms*, which restrict the set of well-formed execution graphs to those observable when executing the program on the respective machine. For instance, when considering the message passing execution graph for x86 in Fig. 1 (bottom left), we observe a $\mathsf{po};\mathsf{rfe};\mathsf{po};\mathsf{fre}$ cycle which is *impossible* on x86, as the x86 model requires $[\mathsf{W};\mathsf{po};[\mathsf{W}] \cup [\mathsf{R}];\mathsf{po};[\mathsf{R}] \cup \mathsf{rfe} \cup \mathsf{fre}$ to be acyclic (among other things) [29]. These restrictions are formally captured by axioms in an architecture's weak memory model [5], which any proof mechanization must thus also include.

Proof objective. To prove a mapping correct, we must prove that *for any* source program P_s that maps to P_t: Any behavior of P_t was also observable for program P_s. In practice, we prove *graph robustness* [19] with Burrow, where we show that for any *execution graph* X_t of target program P_t, there exists an execution X_s of source program P_s with the same final memory state.

Note that graph robustness implies *state robustness* [19, Prop. 4.10], but *not* the other way around. As Burrow was designed to prove graph robustness along a mapping—as is typical for mappings [19,17,4]—(hypothetical) mappings that are state robust but not graph robust cannot be considered by Burrow.

Proof structure. Given a target execution graph X_t, we *construct* the source execution X_s *backwards* along the mapping. For instance, for the example shown in Fig. 1, we are given the Arm program's execution (bottom right) and have to *construct* the corresponding execution of the x86 program (bottom left).

We must then prove that the constructed source execution X_s is well-formed and consistent with the source memory model. For that proof we would map predicates and relations *forward* along the mapping, as we rely on well-formedness and consistency of the target execution X_t to complete the proof.

Case study: x86 to Arm. We use a mapping from x86 to Arm as case study, shown in Fig. 2. This mapping was proposed by the authors of the recent Armv8 model [4, §2.5]. Notably, this mapping was originally subtly incorrect for $\mathsf{CAS}_{\mathsf{AL}}$ (atomic read-modify-write) instructions, but fixed by *changing the Armv8 model* [14,3]; we use that fixed model.

x86		Armv8	
RMOV	(R)	→ LDAPR	(R_{Q})
WMOV	(W)	→ STLR	(W_{L})
MFENCE	(F)	→ F_{SC}	(F_{SC})
RMW	(rmw)	→ $\mathsf{CAS}_{\mathsf{AL}}$	($[\mathsf{R}_{\mathrm{A}}];\mathsf{rmw};[\mathsf{W}_{\mathrm{L}}]$)

Fig. 2: x86 to Arm mapping [4], for instructions and events.

Fig. 1 already showed this mapping for the message passing program, where each x86 store (WMOV) maps to an Arm store-release (STLR), with respective events W and W_{L}; and each x86 load (RMOV) maps to an Arm load-acquirePC (LDAPR), with respective events R and R_{Q}.

2.2 The Agda Proof Assistant

Agda is a dependently typed programming language [2,28], also serving as a proof assistant where Agda programs are proofs of the theorems represented by

the program's types. In contrast to the commonly-used Rocq prover [35], which favors extensive use of *tactics* to solve goals in its internal term language, Agda programs use the terms directly. Idiomatic Agda relies on *dependent pattern matching* [11,28], which Lst. 1 illustrates with a snippet from Burrow.

Lst. 1. Dependent pattern matching in Agda

```
data Event : Set where
  event-r : UniqueId → ThreadId → Location → Value → LabR → Event
  event-w : UniqueId → ThreadId → Location → Value → LabW → Event
  event-f : UniqueId → ThreadId → LabF → Event
  …

data EvR : Pred₀ Event where
  ev-r : EvR (event-r uid tid loc val lab-r)

r-val : {x : Event} → EvR x → Value
r-val {event-r _ _ _ val _} ev-r = val
```

The abstract data type `Event` defines events, of which each inhabitant is either a read (R), denoted by the constructor `event-r`; write (W), denoted by `event-w`; or fence (F), denoted by `event-f`. Although all have a unique identifier (`UniqueId`) and thread (`ThreadId`), only R and W events have a location (`Location`) and value (`Value`). Their respective labels (*e.g.,* `LabR`) are architecture-specific, to distinguish between events variants; for instance, R, R_Q, R_A on Arm. For now, we consider only the x86 R events, whose label is simply denoted by `lab-r`.

The predicate `EvR` asserts that a given event is a R, meaning its inhabitants are always constructor `event-r`, proven by `ev-r`. The function `r-val` returns the value of a R event. Crucially, as `ev-r` asserts that `x` is `event-r`, we need *not* pattern match on the cases `event-w` and `event-f`, as their patterns are empty.

3 Design

Burrow features a modular architecture, depicted in Fig. 3. We additionally show its dependencies on our separate library Dodo for predicates (1) and binary relations (2), containing hundreds of definitions and lemmas we used inside Burrow and which are useful when proving mappings. We separately depict the connection between Burrow and our x86-to-Arm case study to highlight their dependency structure. In the following subsections, we explain the steps of proving a mapping with Burrow.

3.1 Architecture Model Specification

Before proving the mapping itself, we must formalize the weak memory models involved, which Burrow's shared primitives (3) generalize over. For instance, we use those when defining Arm (8) and x86 (9). Examples of shared primitives are `UniqueId` and `ThreadId`, already shown in Lst. 1. As the general structure of

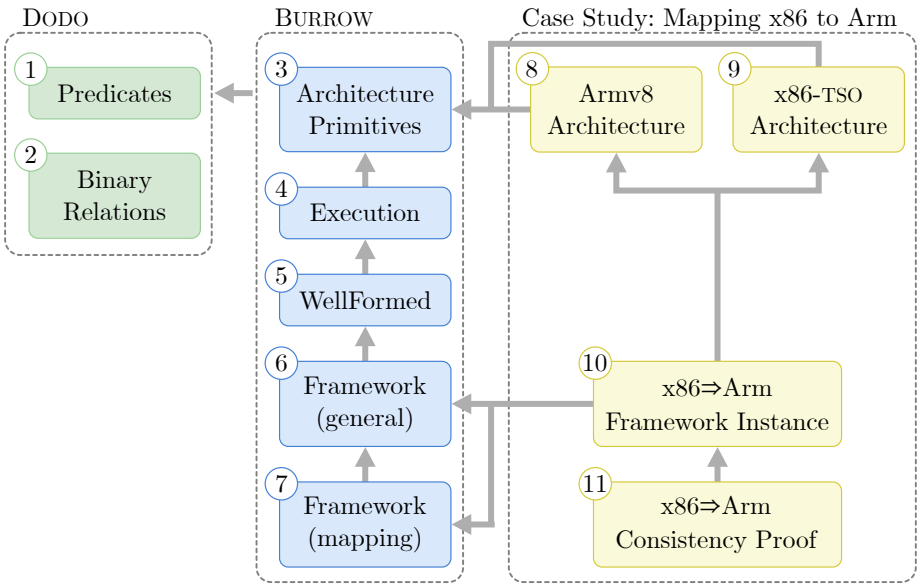

Fig. 3: The architecture of our weak memory proof library Burrow, its dependencies on our relation library Dodo, and its interaction with the x86-to-Arm mapping proof.

events is the same between all models, consisting of either R, W, and F events, we generally define them in Burrow.

Architectures often annotate the base events with different *labels*, for instance, to distinguish between Arm's R, R_A, and R_Q events—all order differently. In contrast, x86 only has regular R events, which order strong by default. Lst. 2 and 3 show mechanizations of R-labels for x86 and Arm, respectively.

Lst. 2. Read Label (x86)

```
data LabR : Set where
  lab-r : Tag → LabR
```

Lst. 3. Read Label (Arm)

```
data LabR : Set where
  lab-r : Tag → LabR -- ^ regular read
  lab-a : Tag → LabR -- ^ acquire read
  lab-q : LabR       -- ^ acquirePC read
```

Semantics vs. program syntax. Mapping proofs consider only the execution graphs generated by the program; the program itself is only considered *implicitly*. However, for some mappings, knowledge about the program is crucial. The x86-to-Arm mapping scheme (Fig. 2) maps x86 W events generated by both `WMOV` and `RMW` instructions to W_L events on Arm. However, the latter orders stronger on Arm [3], but from the event itself, we do not know where it was mapped from. To account for those differences, we explicitly label events with a `Tag`, characterizing them as either being generated by a `RMW` (`trmw`) or a non-`RMW` instruction (`tmov`).

[3] This difference in ordering does not affect the mapping we consider in this paper, but was crucial for our earlier proofs [14,32]. See Risotto [14, §3.3] for details.

Well-formed. By defining the labels for all three base events R, W, and F, we have defined all needed *primitive* components of an execution. Consequently, Burrow then provides the general definition of an execution (4), consisting of those events and the primitive relations po, rf, mo, and rmw, which are shared by most models. Burrow can then define well-formedness (5) of an execution (from § 2.1), which consists of 27 primitive properties in our formalization, with dozens of additional properties derived.

Relations. The *relations* of a model are often *derived* from the primitive events and relations. For instance, x86's *preserved program order* ppo is defined in terms of the po relation, and the R and W events, interpreted as predicates. Lst. 4 shows the Agda formalization of ppo adjacent to its mathematical definition [29].

Lst. 4. x86's ppo relation in Agda

```
data Xppo (x y : EventX86) : Set where
  xppo-w×w : ( (EvW ×₂ EvW) ∩₂ po ) x y → Xppo x y
  xppo-r×w : ( (EvR ×₂ EvW) ∩₂ po ) x y → Xppo x y
  xppo-r×r : ( (EvR ×₂ EvR) ∩₂ po ) x y → Xppo x y
```

$$\begin{aligned} \mathsf{ppo} \triangleq\ & ((\mathsf{W} \times \mathsf{W}) \cap \mathsf{po}) \\ \cup\ & ((\mathsf{R} \times \mathsf{W}) \cap \mathsf{po}) \\ \cup\ & ((\mathsf{R} \times \mathsf{R}) \cap \mathsf{po}) \end{aligned}$$

Not all relations are derived from the primitive relations, which must be separately captured as primitive relations as an add-on to the execution. For instance, Arm's data, ctrl, and addr *dependencies* [4,31] and the mixed-size *same instruction* si relation [13,32] would be primitive in their respective models.

As the general well-formedness definition of Burrow (5) considers only the base primitive definitions, well-formedness axioms of additional primitive relations specific to a given architecture must explicitly be defined. For instance, as we formalize Arm in our case study, which has the above dependencies, we must separately formalize well-formedness for those as part of the Arm architecture definition; an example of which is enforcing $\mathsf{data} \subseteq \mathsf{R} \times \mathsf{W}$.

3.2 Proof: Constructing the Source

After formalizing the models in Burrow, we can prove a mapping between them. Following the proof structure as explained in § 2.1, we are given an execution graph of the target program and *construct* an execution graph of the source. Fig. 1 illustrates the correspondence between two such graphs. For all mapping proofs (but not all optimization proofs), we can obtain the source execution simply my mapping backwards all individual events and primitive relations in the target execution, which Fig. 1 illustrates once with `ev[⇐]`. Lst. 5 shows the partial implementation of `ev[⇐]` in Burrow, only for R events.

We pattern match on the event `x`, and for the `event-r` case (from Lst. 1), the `uid`, `tid`, `loc`, and `val` map trivially. The architecture-specific label `lab` maps backwards along the mapping (Fig. 2). Note that in the forward mapping a non-`RMW` x86 R (`lab-r tmov`) maps to an Arm R_{Q} (`lab-q`), while a `RMW` x86 R (`lab-r trmw`) maps to an Arm R_{A} (`lab-a trmw`).

Lst. 5. Event Mapping (Backward – Arm to x86)

```
ev[⇜] : {x : EventArmv8} → (x∈dst : x ∈ events dst) → EventX86
ev[⇜] {event-r uid tid loc val lab} x∈dst =
  event-r uid tid loc val (r[⇜] lab x∈dst)
  where
  r[⇜] : (lab : Armv8.LabR) → event-r uid tid loc val lab ∈ events dst → X86.LabR
  r[⇜] (lab-a _) x∈dst = lab-r trmw -- RMW (read) ↦ CAS_AL (read)
  r[⇜] lab-q     x∈dst = lab-r tmov -- RMOV ↦ LDAPR
  r[⇜] (lab-r _) x∈dst = ⊥-elim (¬ev-bound dst-ok x∈dst λ()) -- Unreachable
...
```

Notably, the backward mapping is defined only for events that are part of the target execution, captured by the `x∈dst` predicate. '`events dst`' is a predicate in the `dst` execution formalized in Burrow (4); intuitively, the predicate captures the set of events extensionally. This predicate is needed in the body of the final case, which considers an unordered Arm R. As the mapping can never produce an Arm program with R event, its backward mapping is undefined. Only with `x∈dst` can we assert the impossibility of its existence.

Full source execution. Lst. 5 defines only the backward mapping of individual events, not those of relations. However, relations often map trivially along `ev[⇜]`; for instance, when '`po dst`' holds for `x` and `y`, then '`po src`' holds for `ev[⇜] x` and `ev[⇜] y`. The general proof framework (6) automatically derives the backward mappings of various primitive relations, after instantiating the framework with `ev[⇜]`. This proof framework requires only `ev[⇜]` itself, together with various lemmas asserting it satisfies several simple properties; for instance, we require that `ev[⇜]` preserves the `uid` and `tid`.

Not all primitive relations map trivially for any transformation. For instance, while mappings naturally preserve rf and mo along `ev[⇜]`, not all optimizations do (*e.g.,* read-after-read elimination [34]). While the general proof framework (6) can construct the source execution partially (*i.e.* the events and `po`) with multiple well-formedness properties, it cannot prove those properties that are not generally true. However, as mappings between models *does* preserve rf and mo along `ev[⇜]`, Burrow provides another mapping-specific proof framework (7) that fully constructs the source execution.

For our x86-to-Arm mapping, we see (10) in Fig. 3 depends on both frameworks. However, optimization proofs would use only the general framework (6), requiring more lemmas to be proven manually [34,14]. By using the mapping-specific framework, our x86-to-Arm mapping invokes Burrow to automatically produce the *(i)* x86 source execution and *(ii)* its well-formedness.

3.3 Consistency Proof

After defining the x86 source execution and its well-formedness, we must prove the source execution is consistent with its x86 model [29], captured in Fig. 3 by (11). For each x86 axiom, we prove by contradiction that any assumed cycle on x86 would necessarily produce a disallowed cycle in the Arm target execution.

Consistency proofs require the majority of mapping-specific proof engineering effort. Fortunately, we can make those efforts easier with Burrow by defining primitives that map relations from source to target.

Forward mapping. We illustrate the structure of mapping the x86 ppo relation to the Arm bob relation with Burrow in Lst. 6. `xppo[⇒]bob` maps the relation by pattern matching on the constructors of `Xppo` (cf. Lst. 4). The ordering for both cases starting with R is enforced on Arm by mapping it to R_Q for non-`RMW` Rs, and R_A for `RMW` Rs; that predicate we map with `R[⇒]AQ`.

Lst. 6. Predicate and Relation Mapping (Forward – x86 to Arm)

```
R[$⇒]AQ : Pred[$⇒] EvR (EvA ∪₁ EvQ)
R[$⇒]AQ {event-r _ _ _ _ (lab-a _)} x∈dst ev-r = inj₁ ev-a
R[$⇒]AQ {event-r _ _ _ _ lab-q}     x∈dst ev-r = inj₂ ev-q
R[$⇒]AQ {event-r _ _ _ _ (lab-r _)} x∈dst ev-r =
  ⊥-elim (¬ev-bound dst-ok x∈dst λ())

R[⇒]AQ : Pred[⇒] EvR (EvA ∪₁ EvQ)
R[⇒]AQ = [$⇒]→₁[⇒] R[$⇒]AQ

xppo[⇒]bob : Rel[⇒] Xppo Bob
xppo[⇒]bob x∈src y∈src (xppo-r×w ((x-r , y-w) , po[xy])) =
  bob-acq ((refl , R[⇒]AQ x∈src x-r) ⨾[ _ ]⨾ po[⇒] x∈src y∈src po[xy])
xppo[⇒]bob x∈src y∈src (xppo-r×r ((x-r , y-r) , po[xy])) =
  bob-acq ((refl , R[⇒]AQ x∈src x-r) ⨾[ _ ]⨾ po[⇒] x∈src y∈src po[xy])
...
```

The difference between `R[$⇒]AQ` and `R[⇒]AQ` is subtle, but important in Agda. While `ev[⇐]` (cf. Lst. 5) was defined over *target* events, when mapping forward we would want to define properties *over the source* events. However, Agda cannot pattern match on source events, as those are derived from target events by `ev[⇐]`. The fundamental reason is that Agda cannot determine for each source event constructor whether it is a possible *output* of `ev[⇐]` without knowing its inputs; this issue is colloquially known as *green slime* [26]. Instead, `R[$⇒]AQ` must pattern match on the *inputs* of `ev[⇐]`, which are target events satisfying `x∈dst`. However, using well-formedness, Burrow can convert it to the alternative variant `R[⇒]AQ` that takes a source event as input.

Full proof. The full consistency proof involves mapping many such relations from the source (*i.e.* x86) to the target (*i.e.* Arm), for which Burrow provides many types, primitives, and helpers in its frameworks (6) and (7). Lst. 6 showed several types (*e.g.,* `Pred[$⇒]` and `EvR`), primitives (`po[⇒]`), and helpers (*e.g.,* `[$⇒]→₁[⇒]`) provided by Burrow. Beyond those utilities that simplify proving consistency, Burrow fundamentally cannot prove more generally, as those parts of the proof are specific to the involved models, relations, and mappings.

3.4 Case Study Development

We briefly elaborate on the composition of the mechanized proofs for the x86-to-Arm mapping case study. The full proof consists of five components: *(i)* the source execution, *(ii)* well-formedness of that source execution, *(iii)* equivalence of the final memory state of source and target, *(iv)* consistency of the source, and *(v)* the top-level theorem statement for the mapping. Burrow can automatically produce *(i)*, *(ii)*, and *(iii)*, with its frameworks that require only minimal instantiation. Although consistency *(iv)* cannot be proven generally by Burrow, it is easier to prove with Burrow's primitives.

Line counts. We report approximate line counts for our case study and the relevant fragments of Burrow. These numbers are intentionally only approximate because Agda code can be either *(i) verbose but simple*, for instance, when pattern matching on $n \times n$ constructors when capturing propositional equality; or *(ii) compact but complex*, for instance, when a type encodes an invariant or when implicit arguments are absent from the surface syntax, resulting in a small term capturing much complexity. As such, we include these line counts only for conventional completeness.

The line counts for our case study total approximately 1,200 lines of Agda, composed of:

- Burrow framework instantiations: ∼150 lines
- "Glue" code: ∼300 lines
- Consistency proof: ∼750 lines

In addition to those mapping-specific proof components, the fragments of Burrow that the case study relies upon consist of the following approximate number of Agda lines:

- Burrow's well-formedness proof: ∼1,200 lines
- Burrow's primitives and lemmas: ∼700 lines

Without Burrow, those ∼1,200 lines for well-formedness must be written for each mapping (minus some generality overhead), while requiring only ∼150 fewer lines to instantiate Burrow's frameworks; thus totaling ∼2,250 lines, contrasting the ∼1,200 lines currently needed for the mapping proof. Note also that the framework instantiations and glue code are *quick and standard* to write, of which Agda can auto-complete most. Only writing the consistency proof is complex and time-consuming. However, without Burrow, that consistency proof would have needed much more than ∼750 lines, as it heavily relies on Burrow's primitives and lemmas (250+ uses).

4 Related Work

Semantics & proofs. Much work on formal weak memory semantics precedes Burrow. For instance, Alglave *et al.* [5] propose axiomatic semantics for Arm and Power, which their Herd tool extensively tests on Arm and Power hardware.

While they do not write *mapping* proofs, they prove the equivalence of their axiomatic and operational models in Rocq [8]. Their later work introduces the Armv8 model from our case study [4]. Other work proposes formal weak memory semantics for C++ [7], mechanized in Isabelle/HOL [27], but parts of their proofs are only on pen-and-paper. Some of their results were later found incorrect and fixed [20], along with more pen-and-paper proofs.

While prior work often brings significant theoretical results, these are orthogonal to the *mechanization* of such results in a proof assistant, which few solutions address. Notably, the *intermediate memory model* (IMM) [30] has extensive mechanized proofs. IMM reduces the proof burden as an *intermediate* model, requiring fewer proofs between multiple high-level source languages and low-level targets. However, the IMM model is not suitable as an intermediate model for *any* two languages; in particular, for our case study, as both x86 and Arm are stronger than IMM, mapping from x86 to IMM would insert additional unnecessary fences that remain when mapping to Arm. Unlike our Burrow, IMM does not address the mechanization challenges inherent in *writing* those proofs. IMM follows a different proof structure than Burrow as it does not explicitly construct the source execution, but reinterprets the same execution under both source and target models. In addition, IMM's mechanization was developed only for mappings between languages, *not optimizations*. In contrast, we distinguish between the general framework and mapping framework in Burrow, where the former also simplifies proving optimizations correct. Separately, IMM uses their Rocq library *Hahn* [15] with lemmas and tactics for binary relations, mirroring our library Dodo.

Our earlier proofs for binary translation [34,14,32] guided Burrow's development. Unlike the mapping we proved for this paper—as proposed by Alglave et al. [4]—those earlier mapping schemes insert explicit fences instead of using stronger load and store instructions.

Agda vs. Rocq. Many prior mechanized proofs explained above are written in Rocq [30,5,16], contrasting our choice of Agda for Burrow. We elaborate here on the key differences between Agda and Rocq. Idiomatic Rocq favors the use of *tactics*, with which proof terms can be automatically produced. As its surface language is concise, Rocq is often *easier to write*. However, as proof terms do not appear in that surface language, it is *hard to read*, often requiring mentally simulating transformations to the proof state. In contrast, idiomatic Agda represents terms explicitly, while manipulating them without tactics, making proofs often *harder to write*. In practice, Agda's automation is used *while writing* proofs, for instance, to automatically split on cases and to search for simple proof terms. As a consequence, all terms and their manipulation appear explicitly in the Agda source. Finally, Agda proofs rely on *dependent pattern matching* [11,28], making clauses look closer to their mathematical definition while impossible cases disappear. All these characteristics make Agda proofs often *easier to read*.

Tools. ArMOR [24] detects ordering violations on Arm at runtime, but does not provide formal guarantees and cannot capture all semantics [14]. Another work [10] validates that LLVM optimizations for C11 programs do not incorrectly

reorder weak memory primitives. GenMC [18] is a model checker that detects reordering violations by exhaustively exploring a program's state space under weak memory. Kater [17] automatically checks metatheory questions—including mapping correctness—automatically on finite automata capturing weak memory models. While more automatic, Kater's soundness claim depends on its correct implementation, making its claims weaker than mechanized proofs. C11Tester [22] explores many executions to detect violations under weak memory.

5 Conclusion

We propose Burrow, an Agda proof framework for weak memory mapping and transformation proofs. Burrow contains domain-specific abstractions and primitives for weak memory proofs, with which it proves large fragments of weak memory proofs *generally*. We developed Dodo, which is an Agda proof library for predicates and binary relations, independent from weak memory proofs but internally used by Burrow. We demonstrate Burrow's effectiveness by proving a conjectured mapping from x86 to Arm correct. We expect Burrow will also similarly simplify the mechanization of future weak memory proofs.

Data-Availability Statement. Our entire proof development is publicly available and open-source. Our artifact includes Burrow, Dodo, and our x86-to-Arm mapping case study, which is available at: https://doi.org/10.5281/zenodo.19770333

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Adams, M.: Refactoring Proofs with Tactician. In: Bianculli, D., Calinescu, R., Rumpe, B. (eds.) Software Engineering and Formal Methods. pp. 53–67. Springer Berlin Heidelberg, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-49224-6_6
2. Agda Development Team: Agda documentation (2026), https://agda.readthedocs.io/en/stable/
3. Alglave, J.: GitHub Pull Request: "[AArch64 cat] Atomics strengthening". https://github.com/herd/herdtools7/pull/322, merged on 2022-02-11
4. Alglave, J., Deacon, W., Grisenthwaite, R., Hacquard, A., Maranget, L.: Armed Cats: Formal Concurrency Modelling at Arm. ACM Trans. Program. Lang. Syst. **43**(2) (Jul 2021). https://doi.org/10.1145/3458926
5. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data-mining for weak memory. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 40. PLDI '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2594291.2594347

6. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC atomics in C11 and OpenCL. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 634–648. POPL '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837637
7. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 55–66. POPL '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1926385.1926394
8. Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions. Springer Publishing Company, Incorporated, 1st edn. (2010). https://doi.org/10.1007/978-3-662-07964-5
9. Bourke, T., Daum, M., Klein, G., Kolanski, R.: Challenges and Experiences in Managing Large-Scale Proofs. In: Jeuring, J., Campbell, J.A., Carette, J., Dos Reis, G., Sojka, P., Wenzel, M., Sorge, V. (eds.) Intelligent Computer Mathematics. pp. 32–48. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31374-5_3
10. Chakraborty, S., Vafeiadis, V.: Validating optimizations of concurrent C/C++ programs. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization. p. 216–226. CGO '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2854038.2854051
11. Coquand, T.: Pattern matching with dependent types. In: Informal proceedings of Logical Frameworks. vol. 92, pp. 66–79 (1992)
12. Dietrich, D., Whiteside, I., Aspinall, D.: Polar: A Framework for Proof Refactoring. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 776–791. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_52
13. Flur, S., Sarkar, S., Pulte, C., Nienhuis, K., Maranget, L., Gray, K.E., Sezgin, A., Batty, M., Sewell, P.: Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 429–442. POPL '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009839
14. Gouicem, R., Sprokholt, D., Ruehl, J., Rocha, R.C.O., Spink, T., Chakraborty, S., Bhatotia, P.: Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. p. 107–122. ASPLOS 2023, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3567955.3567962
15. Hahn: A Coq library. https://github.com/vafeiadis/hahn
16. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 175–189. POPL '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009850
17. Kokologiannakis, M., Lahav, O., Vafeiadis, V.: Kater: Automating weak memory model metatheory and consistency checking. Proc. ACM Program. Lang. **7**(POPL) (Jan 2023). https://doi.org/10.1145/3571212
18. Kokologiannakis, M., Vafeiadis, V.: GenMC: A Model Checker for Weak Memory Models. In: Computer Aided Verification: 33rd International Conference, CAV

2021, Virtual Event, July 20–23, 2021, Proceedings, Part I. p. 427–440. Springer-Verlag, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-81685-8_20
19. Lahav, O., Margalit, R.: Robustness against release/acquire semantics. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 126–141. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314604
20. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 618–632. PLDI 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062352
21. Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers **C-28**(9), 690–691 (1979). https://doi.org/10.1109/TC.1979.1675439
22. Luo, W., Demsky, B.: C11Tester: a race detector for C/C++ atomics. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. p. 630–646. ASPLOS '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3445814.3446711
23. Lustig, D., Sahasrabuddhe, S., Giroux, O.: A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 257–270. ASPLOS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3297858.3304043
24. Lustig, D., Trippel, C., Pellauer, M., Martonosi, M.: ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures. In: 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA). pp. 388–400 (2015). https://doi.org/10.1145/2749469.2750378
25. Manerkar, Y.A., Trippel, C., Lustig, D., Pellauer, M., Martonosi, M.: Counterexamples and Proof Loophole for the C/C++ to POWER and ARMv7 Trailing-Sync Compiler Mappings (2016), https://arxiv.org/abs/1611.01507
26. McBride, C.: A polynomial testing principle. https://personal.cis.strath.ac.uk/conor.mcbride/PolyTest.pdf (2012)
27. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002). https://doi.org/10.1007/3-540-45949-9
28. Norell, U.: Towards a practical programming language based on dependent type theory, vol. 32. Chalmers University of Technology (2007)
29. Owens, S., Sarkar, S., Sewell, P.: A Better x86 Memory Model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics. pp. 391–407. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_27
30. Podkopaev, A., Lahav, O., Vafeiadis, V.: Bridging the gap between programming languages and hardware weak memory models. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290382
31. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. Proc. ACM Program. Lang. **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158107

32. Reimers, S., Sprokholt, D., Fink, M., Augoustis, T., Kammermeier, S., Rocha, R.C.O., Spink, T., Gouicem, R., Chakraborty, S., Bhatotia, P.: Arancini: A Hybrid Binary Translator for Weak Memory Model Architectures. In: Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. p. 157–174. ASPLOS '26, Association for Computing Machinery, New York, NY, USA (2026). https://doi.org/10.1145/3779212.3790127
33. RISC-V Instruction Set Manual (2024)
34. Rocha, R.C.O., Sprokholt, D., Fink, M., Gouicem, R., Spink, T., Chakraborty, S., Bhatotia, P.: Lasagne: A Static Binary Translator for Weak Memory Model Architectures. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 888–902. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523719
35. Rocq Development Team: Rocq Prover. INRIA (2026), https://rocq-prover.org/
36. Sarkar, S., Memarian, K., Owens, S., Batty, M., Sewell, P., Maranget, L., Alglave, J., Williams, D.: Synchronising C/C++ and POWER. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 311–322. PLDI '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2254064.2254102
37. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 175–186. PLDI '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1993498.1993520