A

Atomicity Refinement for Verified Compilation

Suresh Jagannathan, Purdue University
Vincent Laporte, INRIA Rennes
Gustavo Petri, Purdue University
David Pichardie, INRIA Rennes
Jan Vitek, Purdue University

We consider the verified compilation of high-level managed languages like Java or C# whose intermediate representations provide support for shared-memory synchronization and automatic memory management. Our development is framed in the context of the Total Store Order relaxed memory model. Ensuring complier correctness is challenging because high-level actions are translated into sequences of non-atomic actions with compiler-injected snippets of racy code; the behavior of this code depends not only on the actions of other threads, but also on out-of-order reorderings performed by the processor. A naïve proof of correctness would require reasoning over all possible thread interleavings. In this paper we propose a refinement-based proof methodology that precisely relates concurrent code expressed at different abstraction levels, cognizant throughout of the relaxed memory semantics of the underlying processor. Our technique allows the compiler writer to reason compositionally about the atomicity of low-level concurrent code used to implement managed services. We illustrate our approach with examples taken from the verification of a concurrent garbage collector.

Categories and Subject Descriptors: F.3.1 [**Logics and meanings of programs**]: Specifying and verifying and reasoning about programs, mechanical verification; D.2.4 [**Software engineering**]: Software/program verification, correctness proofs, formal methods, reliability; D.3.4 [**Programming languages**]: Processors, compilers, optimization; D.1.3 [**Concurrent Programming**]: Parallel programming

General Terms: Languages, Reliability, Security, Verification

Additional Key Words and Phrases: Verified compilation, managed languages, concurrency, garbage collection, compiler transformations and optimizations, refinement, atomicity, mechanized proof assistant (Coq)

ACM Reference Format:
ACM Trans. Embedd. Comput. Syst. V, N, Article A (January YYYY), 30 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Managed languages provide intrinsic support for concurrency at several levels. Applications can express concurrent computations using threads and synchronization primitives. Additionally, to improve scalability or performance, elements of the language implementation itself may run concurrently with application threads. The interactions between application threads and the language runtime system such as the garbage collector are regulated by compiler-injected code snippets. Example of snippets include allocation fast paths, read and write barriers, synchronization fences, data initialization checks. The code injected by the compiler is sophisticated, often racy, and must operate correctly in the presence of program transformations, both local and global. The complexities of modern language runtimes justify the effort of verified compilation.

However, verifying the correctness of a compiler for languages such as Java or C# is an ambitious goal. It entails reasoning about the inherently parallel behavior of operations in the source language, as well as the operations introduced by the compiler. Consider a `synchronized` block in Java; it is translated to a pair of `monitorenter` and `monitorexit` operations in the bytecode. Acquiring a lock to implement `monitorenter` can be *modeled* as an atomic action that checks the lock's availability and transfers ownership if the lock is unowned. The actual *implementation* uses a spin-lock with the following sequence of operations:[1]

```
repeat
  old <- CAS[Lock, 0 -> 1];
  current = old;
  while (current != 0) do current <- Lock;
until (old == 0)
```

The code may be interleaved with other threads. The loop first attempts a compare-and-set to acquire the lock: a `CAS[X,e->n]` statement atomically reads `X`, compares its value to the expression `e` and, if they are equal, writes `n` into it; it always returns the read value. If the lock was not held, control exits the loop. Otherwise, the loop will retry as long as the lock is held.

An abstract definition that is closer to the bytecode semantics could be expressed as follows. Consider this a specification of the low-level implementation:

```
atomic(assume(Lock == 0); Lock := 1)
```

In the above snippet, `atomic` executes *only if* `assume` holds. This specification guarantees that no action by other threads on `Lock` can take place between the assume statement and the assignment. While easier to reason about, it is impractical to implement. So, the low-level version is efficient, but proving it implements `monitorenter` is difficult. Clearly, proving any invariant would be easier using the high-level specification, since there are fewer interleavings to consider. In other words, the low-level implementation exposes fine-grained behavior, while the high-level specification reflects a coarser granularity. We propose to mitigate the dichotomy between low-level implementations, and high-level specifications by establishing a *refinement* predicate that relates the "high" and "low" definitions of concurrent code. Informally, we say that a low-level statement l *refines* a high-level one h if the execution of both l and h starting from the same state leads to the same final state; furthermore, if executing l admits a trace tr of interleaved actions of other threads, then tr must be admissible as a feasible trace under the execution of h. This notion of refinement guarantees the equivalence of high and low-level code. Given a high-level specification h that captures the atomicity properties implicit in l, the refinement predicate helps the compiler writer devise a proof that l *refines* h.

Verification Challenge. While recent years have seen progress in compiler verification, much of this work has been for sequential languages [Leroy 2009]. The basic correctness argument requires proving that any behavior admitted by the compiled program is also admitted by the source. This is typically shown by a *backward* simulation proof between target and source language semantics. Assuming the source program is safe, a backward simulation demonstrates that any observable behavior produced by the target program is a valid observable behavior of the source program as defined

[1] Capitalized variables refer to objects allocated in shared memory, lowercase variables refer to thread-local objects or registers, '=' is a local or register move, '<-' loads from a shared variable, and ':=' stores into a shared variable.

by the source language semantics [Leroy 2009]. Demonstrating such a simulation is complicated by the presence of concurrency. While there are success stories [Ševčík et al. 2011; Leroy et al. 2012], these efforts have focused on a subset of C [Leroy 2009]. A fundamental characteristic of the C language is that individual memory accesses performed by source programs are compiled into individual memory accesses at the low-level target. This property makes a standard simulation argument tractable. On the other hand, managed languages often compile a single source memory access to multiple low-level memory accesses, as a result of code *injected* by the compiler. For example, Java compilers typically inject write barriers before each field update to support garbage collection. Indeed, an implementation of write barriers, such as the one defined by the DLG collector [Doligez and Leroy 1993], uses a non-trivial protocol to communicate with the garbage collector thread, and serves to notify that changes are being done in the object graph. Dealing with concurrency is thus quite challenging since it requires proving concurrent invariants of the underlying implementation of the compiler and runtime system, internal data structures, and communication protocols. The details of these protocols are not visible to the high-level source. Consequently, a naïve approach to verification of injected concurrent code fragments is not scalable using a standard backward simulation argument.

Atomicity Refinement. To address this challenge, we propose an atomicity refinement methodology that coarsens the granularity of injected code, therefore simplifying the overall verification of the compiler infrastructure. Our approach facilitates the modular expression of such proofs, making a backward simulation argument feasible by establishing the equivalence of fine-grained and coarse-grained representations of concurrency operations, in isolation of the other components in the program. Our refinement enables a simulation argument similar to the ones used to demonstrate the correctness of sequential optimizations, and hence allows such arguments to be effectively applied to potentially racy, lock-free, concurrent code. Our approach is motivated by the premise that establishing that the high-level specification captures the behavior defined by the source program is substantially easier than directly proving the correspondence between low-level target code and source code.

Relaxed Memory. A fundamental premise of our development is that performance must not be affected. Indeed, if this was not a concern we could use the DRF-0 [Adve and Hill 1990] programming discipline, which guarantees Sequential Consistency (SC) in the absence of data races. However, disallowing data races requires synchronizing access to share data. This would have significant costs. Consider the example of a write barrier injected by a Java compiler before every memory update operation. Imposing synchronization on those barriers would mean that every write in the source Java program would be preceded by a synchronization. All productions Java compilers emit racy code in barrier, it is therefore mandatory to consider the underlying relaxed memory model of the architecture. There are techniques to relate programs with different atomicity granularities in the context of an SC memory model [Elmas et al. 2009; Turon and Wand 2011; Liang et al. 2012a]. However, imposing SC semantics on executions operating within a relaxed memory setting such as Total Store Order (TSO), would either be erroneous or impact performance. To illustrate, consider Dekker's algorithm, a technique to enforce mutual exclusion between two threads. Here $\texttt{i} \in \{0, 1\}$ represents the thread identifier.

```
Flag[i] := true;
atomic ( while (Flag[1 - i]) do skip;
         Critical Section );
Flag[i] := false
```

Assuming SC, existing techniques can verify that an implementation without the atomic block refines one that includes it. This is incorrect under TSO since stores may be buffered: thus, the update to `Flag[i]` performed by one thread may not be visible to the read of `Flag[1-i]` by the other. Techniques proved sound for SC are not sufficient when dealing with TSO-like architectures. One could imagine imposing SC on programs running in a TSO-like architecture by adding fence instructions after each store. However, this would degrade performance. The TSO memory model supports programming idioms based on *publication* where a shared variable is used as a flag to indicate that some changes have been made observable. The following is a canonical example (ignoring the commented annotations for an instant):

```
Data := ...; // @Local          ||   while (! Flag) do skip; r <- Data; //
Flag := true                         @Local
```
(1)

The code uses the shared variable `Flag` to communicate the availability of `Data` across threads. Although the idiom contains a data race on `Flag`, the read of `Data` is guaranteed to witness the update done by the other thread. No synchronization or fence required. Of course, if a third thread was to modify both `Data` and `Flag`, the guarantee could be violated. This shows that we must be able to capture invariants about the behavior of the *environment*. Our methodology allows injected code to be annotated with *rely* conditions [Vafeiadis and Parkinson 2007; Dodds et al. 2009]; the @Local annotations declare assumptions that the memory location being stored (resp. being read) will not be read or written by the environment (resp. will not be written). In the example both threads respect the @Local annotations through the publication protocol. These annotations have no runtime effect; they serve to enable refinements that would otherwise not be sound. In particular, if we know that no other thread can observe an intermediate state, one can soundly assume that the subsequent instructions are atomic with respect to the environment, and therefore can be refined to a coarser atomic block. Hence, annotations enable "thread-modular" reasoning about the global interferences of the system. Section 5 explains how to prove their correctness.

Contribution. We present a proof methodology to verify the correctness of compiler translations from a high-level Java-like Intermediate Representation (IR) with object allocation, field access, thread creation and synchronization to a low-level structured Register Transfer Language (RTL) representation expressed in an IR called $\text{RTL}^{\mathcal{I}}$. The $\text{RTL}^{\mathcal{I}}$ IR is patterned after CompCertTSO's [Ševčík et al. 2013] RTL, an IR that expresses unstructured control flow graphs, additionally allowing the expression of high- and low-level statements; these statements are expressed in a structured language called $\mathcal{I}$. Our methodology is based on an expressive notion of refinement that enables lightweight compositional reasoning of concurrent and racy code within a verified compiler framework. We concentrate on the code that is injected by the compiler to support services such as allocators, collectors, synchronization, etc.

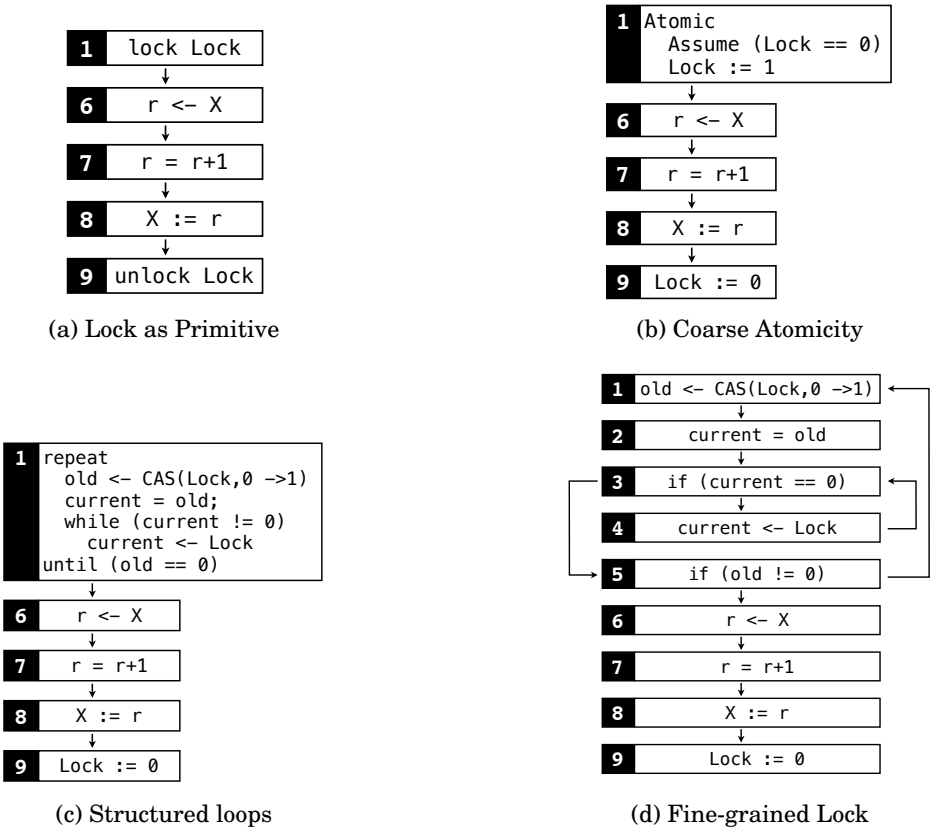
Our work is part of a larger effort to verify a realistic Java implementation integrated within the CompCertTSO verified compiler stack [Ševčík et al. 2013]. The refinement technique supports TSO relaxed memory semantics to allow the verification of low-level concurrent code in the context of x86 multiprocessors.

The remainder of the paper is structured as follows. In the next section, we provide a motivating example. Section 3 gives an overview of our approach. Section 3.1 describes the refinement rules that enable the sound transformation from low-level code to high-level specifications. Section 4 discusses uses and benefits of the refinement methodology to verify the translation of the major components in a concurrent garbage collector. Section 5 formalizes the approach. Related work and conclusions are given in Sections 6 and 7.

2. MOTIVATION

We illustrate our approach with a program that atomically increments a shared variable. We start from the Java-like source language of Figure 1a and aim to compile this code to the RTL program of Figure 1d where the abstract locking operations are fully explicited. To directly verify that the RTL code only admits the behaviors permitted by the source program is challenging given the inherently racy execution of the statements responsible for acquiring the lock. As an example, consider the simulation between source and target programs shown in Figure 2. There we have three threads executing the same RTL code. The simulation is clearly complex, with alignment of source and target states often skewed by many intermediate steps. Our methodology allows these states to be more cleanly and straightforwardly aligned by introducing two extra translation steps: a high-level specification shown in Figure 1b and a low-level implementation shown in Figure 1c. The low-level implementation expresses the implementation of atomic actions in term of low-level atomic memory operations, and structured control-flow via loop and repeat statements. Significantly, Figure 1c shows that statement (1) is isolated from the other operations in the program. We use this IR as our compilation target.

Our refinement methodology systematically "lifts" low-level code to its higher-level counterpart. Since the low-level includes loops with complex bodies, lifting is per-



(a) Lock as Primitive

(b) Coarse Atomicity

(c) Structured loops

(d) Fine-grained Lock

Fig. 1: Atomic increment of a shared variable.

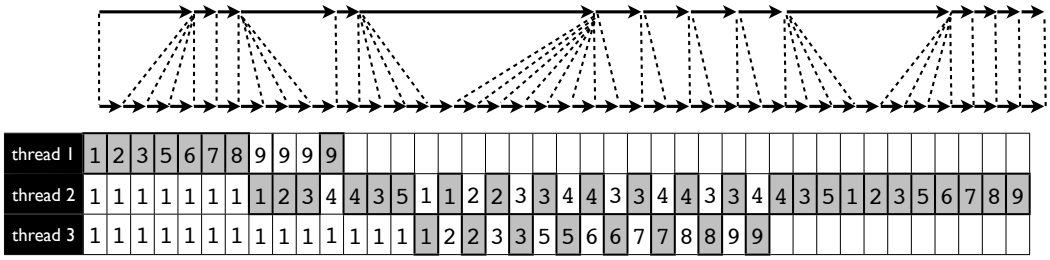


Fig. 2: A backward simulation must match and align program states under source and target semantics. The numbers refer to statements in the RTL program of Figure 1d.

formed systematically, with each step accompanied by a proof of equivalence, along with compositional rules (the core of our contribution) that validate their aggregation. Once this equivalence is established, the low-level non-atomic code fragment can be replaced by the high-level atomic version. To illustrate, consider statement (1). Using our refinement predicate, we can express the outer `repeat` statement in terms of higher-level unconditional `loop` and `assume` operations:

```
loop {
    old <- CAS[Lock, 0 -> 1];
    current = old;
    while (current != 0) do current <- Lock;
    assume(old != 0)
}
old <- CAS[Lock, 0 -> 1];
current = old;
while (current != 0) do current <- Lock;
assume(old == 0);
```

The `loop` construct captures the notion of acquisition failure. The statements after `loop` capture the notion of success; an acquisition succeeds if the value returned by the `CAS` operation indicates the lock was unheld when the operation was performed. Observe that as long as the `loop` *eventually* terminates (i.e., as long as the lock is finally acquired), the actions performed in the loop are irrelevant to understanding the correctness of the code with respect to the high-level definition. In other words, if we consider possible iterations of this loop, the only ones able to match with the high-level executions are those in which the loop does not execute.

Our methodology provides a set of meaning-preserving transformation rules that allow us to simplify the program to eliminate actions not reflected in the specification. Consider the two `assume` statements. Any execution in which these assertions hold is equivalent to executions of a program in which they are lifted above independent statements. This means we can move the `assume` statements to immediately follow their respective `CAS` operations that set the value of `old`:

```
loop {
    old <- CAS[Lock, 0 -> 1]; assume(old != 0);
    current = old;
    while (current != 0) do current <- Lock
}
old <- CAS[Lock, 0 -> 1]; assume(old == 0);
current = old;
while (current != 0) do current <- Lock
```

Regardless of the actions of other threads that may concurrently manipulate Lock, the validity of the assume positioned immediately after the CAS must be the same as its validity in the original example. The resulting specification contains two interesting CAS patterns that can be further simplified using compositional rules available to the compiler writer. First, a sequence

```
x <- CAS(Lock, old -> new); assume(x != old)
```

can be replaced by: fence; x <- Lock. Since the assumption holds precisely when Lock is held by another thread, the effect of the CAS operation is guaranteed to be a simple load of the current value of Lock after flushing the store buffer; as before, this transformation only involves local reasoning, and is not affected by the actions of other threads, which are guaranteed in any successful execution to not violate the conditions checked by the assumption. Second, a successful lock acquisition:

```
x <- CAS(Lock, old -> new); assume(x == old)
```

can be replaced by

```
atomic { x <- Lock; assume (x==old); Lock := new }
```

We anticipate here that the atomic statements implicitly have a TSO fencing semantics at the end. The reasoning that establishes the correctness of this transformation is as before: if the assumption holds, which it must for any successful execution, then the CAS operation must have succeeded, and the value of Lock must be new. Since the CAS operation executes a read-modify-write operation atomically, we can expose the sequence explicitly. We thus obtain

```
loop {
    fence; old <- Lock;
    current = old;
    while (current != 0) do current <- Lock
}
atomic { old <- Lock;  assume(old==0); Lock := 1 }
current = old;
while (current != 0) do current <- Lock;
```

This specification can, in turn, be further simplified because the loop block only modifies dead variables (i.e., local variables that are never used before being redefined) and the last two lines only modify a variable that is not used subsequently. The fence action is clearly dominated by the following atomic which also makes it dead code.

```
atomic { old <- Lock; assume(old==0); Lock := 1 }
```

This specification captures the behavior of all executions in which a lock that is initially unheld is acquired by the executing thread. By simply eliminating the use of the local variable old, we get the high-level specfication found at statement (1) in Figure 1b:

```
atomic { assume(Lock == 0); Lock := 1 }
```

In the context of our example, this equivalence establishes the correctness of the translation of statement (1) in Figure 1c with the specification found in statement (1) of Figure 1b. Assuming the specification is correct, the task of constructing a backward simulation proof that the target program admits the behaviors allowed by the source is significantly simpler given the semantic closeness of the specification (Fig. 1b) to the source (Fig. 1a) and the structured RTL (Fig. 1c) to the unstructured target (Fig. 1d).

3. A REFINEMENT-BASED PROOF METHODOLOGY

We base our refinement methodology around the intermediate language $\mathcal{I}$ (read "Inject"). The pieces of code that need to be injected to the source as part of the compilation, as well as other parts of the runtime system, are written in $\mathcal{I}$. For instance, the garbage collector, and the write barriers that are attached to each memory update, are written in this language. An important aspect of $\mathcal{I}$ is its support for coarse-grained atomic instructions, that while not directly available in the target architecture, are only used to support our atomicity refinement proofs. As such, there is a sublanguage of $\mathcal{I}$ which contains all the low-level (fine-grained) statements that are directly supported by the architecture, we denote this language by $\mathcal{I}_{\mathsf{L}}$ (read "Inject Low").

LANGUAGE : $\mathcal{I}$

$$
\begin{array}{rcl}
s \in \mathcal{I} & ::= & \mathsf{skip} \mid d = op(\vec{r}) \mid s; s \mid \mathsf{if}\ c\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{repeat}\ s\ \mathsf{until}\ c \\
& \mid & \mathsf{load}_{\nu}(d, r) \mid \mathsf{store}_{\nu}(d, r) \mid \mathsf{cas}(d, r, o, n) \mid \mathsf{fence} \mid \mathsf{abort} \\
& \mid & \mathsf{atomic}\ s \mid \mathsf{assume}\ c \mid \mathsf{branch}\ s,\ s \mid \mathsf{loop}\ s \\
\nu & ::= & \mathsf{Local} \mid \epsilon
\end{array}
$$

Fig. 3: Syntax of $\mathcal{I}$.

$$
\text{Source IR} \quad \hookleftarrow \quad \boxed{\mathrm{RTL}^{\mathcal{I}} \quad \Leftarrow \quad \mathrm{RTL}^{\mathcal{I}}_{\mathsf{L}}} \quad \hookleftarrow \quad \text{RTL}
$$

Fig. 4: Proof strategy. The shaded portion is enabled via our refinement methodology. $\mathrm{RTL}^{\mathcal{I}}$ programs are successively refined to replace low-level statements with high-level ones based on our refinement rules. $\hookleftarrow$ is the basic backward simulation. $\Leftarrow$ is the backward simulation from refinement.

Figure 3 presents the $\mathcal{I}$ language, with $\mathcal{I}_{\mathsf{L}}$ restricted to the two first lines of the grammar. $\mathcal{I}_{\mathsf{L}}$ has mostly standard commands with the exception that all statements operate on registers, here ranged by the metavariables d, r, o, n and $\vec{r}$ representing a sequence of registers. $\mathcal{I}_{\mathsf{L}}$ includes skip, sequencing, standard arithmetic and boolean operators, conditionals, repeat−until loops, loads-from and stores-to memory (where the registers are assumed to contain memory locations), a compare-and-set statement corresponding to the CAS instruction found on x86 processors, a fence command for memory ordering purposes, and an abort command to denote exceptional behavior. Notice that the commands $\mathsf{load}_{\nu}(d, r)$ and $\mathsf{store}_{\nu}(d, r)$ have a visibility annotation ν which can be Local or empty. This annotation, which has no runtime effect, indicates in the program syntax that no other thread in the system can modify the references being accessed by the command. For presentation reasons, in the code snippets we present in the figures of the paper we will use commentaries of the form "// @Local" to highlight Local memory accesses. We will describe the usage of these annotations in the refinement rules below. More unusual are the "high-level" assume, branch, loop and the coarse-grained atomic statements which complete the $\mathcal{I}$ language. Atomic statements execute disallowing actions from other threads, loops execute their body a non-deterministic number of times, and branches non-deterministically choose the branch they should execute; "incorrect" choices simply manifest as failed assumptions (expressed through assume) in the resulting execution.

We inject terms of $\mathcal{I}$ on top of the RTL intermediate representation of the CompCertTSO [Ševčík et al. 2011] verified compiler. Thus, some nodes of the RTL language

of CompCertTSO will contain $\mathcal{I}$ statements. $\text{RTL}^{\mathcal{I}}$ (read "RTL-Inject") is the language resulting from combining RTL with $\mathcal{I}$. The code in Figures 1b and 1c are samples of $\text{RTL}^{\mathcal{I}}$ where the first node contain $\mathcal{I}$ code. The sublanguage that results by combining RTL with the $\mathcal{I}_{\mathsf{L}}$ sublanguage of $\mathcal{I}$ is denoted $\text{RTL}^{\mathcal{I}}_{\mathsf{L}}$.

As shown in Figure 4, our refinement methodology is used to systematically replace low-level statements in the $\text{RTL}^{\mathcal{I}}_{\mathsf{L}}$ program (e.g., cas and repeat statements) with the high-level statements (e.g., atomic and assume) of $\text{RTL}^{\mathcal{I}}$. Figure 1 is an example of this process.

3.1. Refinement Rules

Given a low-level statement s_l defined as part of the translation, we must construct a high-level statement that matches a provided specification s_h, defined in terms of atomic, assume, loop, branch, and sequence commands; and a proof that s_l refines s_h (written $s_l \preccurlyeq s_h$). s_l is a proper implementation of s_h whenever the visibility annotations of s_l hold. (We provide a formal definition of this relation in Sec. 5.3.) To ease the construction of such proofs, we provide a set of compositional rules that can be applied interactively using the Coq proof assistant. These rules avoid the need to modify the semantics of any intermediate representation. We show an excerpt of selected rules provided in our development in Figure 5.

The rule TRANS establishes the obvious transitivity property of refinement. IFBRANCH and REPEAT allow control structures to be replaced by a combination of assume, loop and branch statements. For example, a `repeat` statement can be refined into one that executes its body a non-deterministic number of times, verifying that the terminating condition is not satisfied, and a terminating iteration where the condition is satisfied. IFATOMIC allows an `if` whose branches are atomic to be transformed into an atomic `if`.

The CAS-FAIL rule establishes a refinement between a failed CAS operation and a `load` operation that reads the contents of the location in register r into the destination register d. As in x86-TSO, the load performed by the CAS must be preceded by a fence command. A CAS fails when the presumed old value is not the same as the value read. Thus, the sequence of low-level statements that performs the CAS and then assumes the failing condition is a refinement of a simple load on the location. In contrast, a successful CAS must atomically store the new value into the location, assuming the location still contains the presumed old value (CAS-SUCCESS). Notice that unlike CAS-FAIL, the CAS-SUCCESS rule does not require a fence. This is because the semantics of atomic blocks implicitly requires that the TSO write-buffers be empty, similar to the fence instruction (see Section 5). Rule SWAPASSUME lifts assumptions above other statements in a sequence. Rule DEAD allows to remove a statement with an unused effect. This is a typical exercise with racy algorithms: a `while` or `repeat` loop spins until the current thread takes its turn on a shared memory access. By turning such a loop into a mix of loop and assume statements, the last iteration where the thread gets its launching window becomes explicit. The previous iteration block is generally a dead block that can be removed since the actions performed within those iterations have no observable effect. The rule FENCEATOMIC is an obvious consequence of the fencing behavior of atomic that flushes the store buffer upon completion. FENCEELIM allows us to remove unnecessary fences. AFTERABORT indicates that no commands are executed after an abort.

The rule MAKESTOREATOMIC is implied by the fencing behavior of atomic and observing that stores are indivisible operations. A similar argument is applied for MAKELOADATOMIC, but in this case the fence is required to precede the load, which in TSO disallows the load from overtaking previously issued writes in the buffer. Perhaps

$$\textsc{Refl}\ \frac{}{s \preccurlyeq s} \qquad \textsc{Trans}\ \frac{s_1 \preccurlyeq s_2 \quad s_2 \preccurlyeq s_3}{s_1 \preccurlyeq s_3} \qquad \textsc{Repeat}\ \frac{}{\mathsf{repeat}\, s\, \mathsf{until}\, c \preccurlyeq \mathsf{loop}\,(s; \mathsf{assume}\, \neg c\,); s; \mathsf{assume}\, c}$$

$$\textsc{IfBranch}\ \frac{}{\mathsf{if}\, c\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2 \preccurlyeq \mathsf{branch}\,(\mathsf{assume}\, c\,; s_1),\ (\mathsf{assume}\, \neg c\,; s_2)}$$

$$\textsc{IfAtomic}\ \frac{}{\mathsf{if}\, c\, \mathsf{then}\,(\mathsf{atomic}\, s_0)\, \mathsf{else}\,(\mathsf{atomic}\, s_1) \preccurlyeq \mathsf{atomic}\,(\mathsf{if}\, c\, \mathsf{then}\, s_0\, \mathsf{else}\, s_1)}$$

$$\textsc{CAS-fail}\ \frac{}{\mathsf{cas}(d, r, o, n); \mathsf{assume}\, o \neq d \preccurlyeq \mathsf{fence}; \mathsf{load}\,(d, r)}$$

$$\textsc{CAS-success}\ \frac{}{\mathsf{cas}(d, r, o, n); \mathsf{assume}\, o = d \preccurlyeq \mathsf{atomic}\,(\mathsf{load}\,(d, r); \mathsf{assume}\, o = d\,; \mathsf{store}(r, n))}$$

$$\textsc{SwapAssume}\ \frac{\mathsf{defines}(s) \cap \mathsf{uses}(c) = \emptyset}{s; \mathsf{assume}\, c \preccurlyeq \mathsf{assume}\, c\,; s} \qquad \textsc{DeadCode}\ \frac{s_1 \text{ is dead}}{s_1; s_2 \preccurlyeq s_2} \qquad \textsc{FenceAtomic}\ \frac{}{\mathsf{fence} \preccurlyeq \mathsf{atomic}\, \mathsf{skip}} \qquad \textsc{FenceElim}\ \frac{}{\mathsf{fence} \preccurlyeq \mathsf{skip}}$$

$$\textsc{AfterAbort}\ \frac{}{\mathsf{abort}; s \preccurlyeq \mathsf{abort}} \qquad \textsc{MakeStoreAtomic}\ \frac{}{\mathsf{store}\,(d, r); \mathsf{fence} \preccurlyeq \mathsf{atomic}\, \mathsf{store}\,(d, r)} \qquad \textsc{MakeLoadAtomic}\ \frac{}{\mathsf{fence}; \mathsf{load}\,(r, d) \preccurlyeq \mathsf{atomic}\, \mathsf{load}\,(r, d)}$$

$$\textsc{GrowAtomicLocal}\ \frac{s_0 \in \{\ \mathsf{store}_{\mathsf{Local}}(d, r), \mathsf{load}_{\mathsf{Local}}(d, r)\ \}}{s_0; \mathsf{atomic}\, s_1 \preccurlyeq \mathsf{atomic}\,(s_0; s_1)} \qquad \textsc{EFLeft}\ \frac{s_1 \text{ is effect free}}{s_1; \mathsf{atomic}\, s_2 \preccurlyeq \mathsf{atomic}\,(s_1; s_2)}$$

$$\textsc{EFRight}\ \frac{s_2 \text{ is effect free}}{\mathsf{atomic}\, s_1; s_2 \preccurlyeq \mathsf{atomic}\,(s_1; s_2)}$$

Fig. 5: Compositional rules of the refinement predicate (excerpt).

the most interesting rule is GROWATOMICLOCAL which allows local memory operations (i.e., loads and stores) to be moved within an atomic block; such aggregation is clearly acceptable since the effect of the operation is not observable to the environment. This is guaranteed by the Local visibility annotation, which implies that the pointer in the register r cannot be changed by the environment (neither can it be observed in the case of a store). Similar rules EFLEFT and EFRIGHT apply for *effect free* operations (i.e., which only manipulate registers).

To illustrate these rules, consider the spin lock of Section 2, in order to refine that code the following rule are applied: REPEAT, SWAPASSUME, CAS-FAIL, DEAD, FENCEELIM and CAS-SUCCESS. (We treat `while` as syntactic sugar for `repeat`.) This sequence of refinements directly mirrors our informal explanation.

Let us at this point digress and consider the impact of TSO. As it can readily be seen from MAKESTOREATOMIC, MAKELOADATOMIC and GROWATOMICLOCAL, the semantics of TSO implicitly affect the kind of refinement we perform. In particular, GROWATOMICLOCAL, for the case where s_0 is a local store, reflects the publication idiom. To see how it exploits TSO, reconsider the publication example of Equation 1 under a slightly weaker memory model: Partial Store Ordering (PSO) [SPARC 1994].

PSO adds to the write-read relaxation of TSO, the possibility for two writes on different locations to be reordered. Hence, the publication idiom does not hold for PSO. Our GROWATOMICLOCAL rule would be unsound for PSO. Consider the case where we apply GROWATOMICLOCAL to Equation 1 with PSO. The reason why variable `Data` can be considered "Local" is because no other thread can access it before the write of `Flag`. However, in PSO, the value of `Flag` may be propagated to other threads before the write of `Data`. Hence, the read of `Data` might obtain an outdated value, breaking the illusion of an atomic assignment of `Data` and `Flag`. Adding a `fence` instruction before the atomic command in GROWATOMICLOCAL would recover soundness. So, TSO permeate our refinement rules, and changing the memory model to consider weaker memory models would require reconsidering these rules. Similarly, using our rules with SC would impose unnecessary fences on code that needs to be refined.

We end by noting that the rules of Figure 5 are purely syntactical. This helps us reduce the burden of interactively applying them by a set of custom Coq tactics that automatically explore a program tree in order to find a subterm that fits with a given refinement rule. Some rules such as DEAD require discharging some preconditions in order to be applied. We discharge these preconditions using Coq's reflection capabilities; the predicates are executable and we let Coq prove them by computation. Significantly, these rules are sound with respect to the semantics given in Section 5.

4. VALIDATION

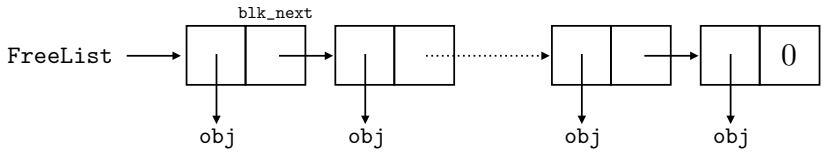
To illustrate the use of our refinement methodology for the verification of a managed concurrent programming language such as Java, we have devised a block-structured Managed Intermediate Representation (MIR), which we compile to $\text{RTL}_{\text{L}}^{\mathcal{I}}$ and subsequently to x86-TSO using the CompCertTSO tool chain. MIR exposes typical features found in a managed language such as object allocation, field access, synchronization, as well as high-level concurrency primitives such as locks, threads, non-blocking stacks and garbage collection. MIR has been designed to serve as a reasonable IR target for Java bytecodes. Our compiler is sufficiently complete to compile data-allocation intensive programs such as the *binary-trees* benchmark.[2] Running this program shows that the collector effectively traces the heap and collects free objects in parallel with user code.

4.1. Case Study: Concurrent Allocation

As a representative case study, we examine the code snippet used to implement a concurrent allocator that interacts with the DLG collector [Domani et al. 2000; Doligez and Leroy 1993; Doligez and Gonthier 1994]. DLG is a *mark and sweep* algorithm with three colors: *White*, *Gray* or *Black*. Objects that are not known to be alive are marked *White*. Thus, every collection cycle starts with all objects white. Objects discovered to be potentially alive, either by the collector or a mutator (a user thread), are upgraded to *Gray*, indicating that the object and its descendants should not be collected in the current cycle. When the collector *traces* the object graph, it marks *Gray* objects *Black* and all of its descendants *Gray*. The traversing of the object graph terminates when no gray objects exist. At this point white objects are known to be unreachable and their memory can be reclaimed in one sweep.

Free memory shared by the allocator and collector is implemented as a Treiber stack [Treiber 1986] of free objects, where for simplicity, in our development, all objects are of the same fixed size. Allocation then is implemented as a pop in the free object stack. The fundamental property of the implementation is that allocation is non-blocking.

[2] http://shootout.alioth.debian.org/



(a) FreeList Data Structure

```
curr <- FreeList;
repeat {
  head = curr;
  If (head == 0) Then abort;
  next <- head[blk_next];    // @Local
  curr <- CAS[FreeList, head -> next]
} until (curr == head);
head[blk_hdr] := header;     // @Local
head[blk_color] := color;
fence;
```

(b) Low

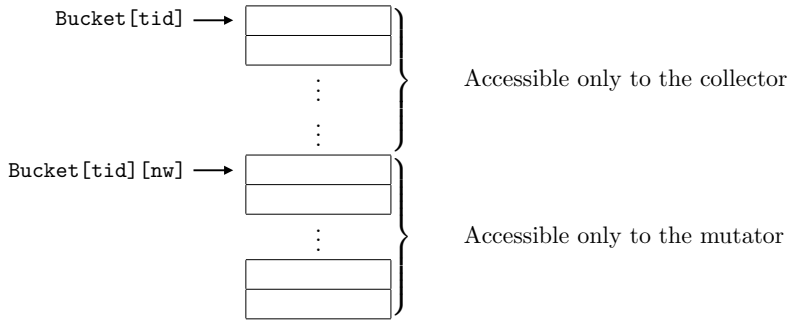
```
branch
  { abort },
  { atomic {
      head <- FreeList;
      assume (head != 0);
      next <- head[blk_next];
      FreeList := next };
    atomic {
      head[blk_hdr] := header;
      head[blk_color] := color  } }
```

(c) High

Fig. 6: A concurrent allocator; GROWATOMICLOCAL, AFTERABORT, CAS-SUCCESS, SWAPASSUME, CAS-FAIL, DEAD and MAKESTOREATOMIC are the salient refinement rules necessary to effect the transformation. For the sake of readability, we use syntactic sugar for load and store operations to operate over simple expressions, rather than just registers, and provide visibility annotations as comments.

We provide the low-level and high-level code of the allocator in Figure 6. A pictorial view of the data structure is presented in Figure 6a. A thread starts the allocation by reading the pointer `FreeList` which points to the first free object in the free stack. Then, it checks if the head is null (`head == 0`). Otherwise, it reads the pointer to the next element of the stack into `next`. Finally, it atomically (with a CAS) verifies whether the head of the stack changed, and in case it did not, makes `FreeList` point to it, effectively removing the head of the stack. If the CAS fails, indicating that another allocation succeeded in between the read of the head and the CAS, the allocator restarts the whole process. Notice that when the final CAS succeeds, it means that no other thread modified the stack in between the read of the head and the CAS, which in turn means that the whole execution can be considered to happen atomically. This is indeed the specification given to the algorithm in Figure 6c. Notice at this point that the field `head[blk_next]` has a `@Local` annotation. This annotation is guaranteed to be true by an invariant of the garbage collector. The only case in which the `blk_next` field is modified is when the collector sweeps unreachable objects. To that end, the collector creates a linked list of the dead objects (modifying their `blk_next` field), and when all dead objects have been put in that list, it performs a concatenation (c.f. push in a Treiber stack) which only modifies the `FreeList` variable. Hence, only the collector is able to modify `blk_next`, and it does so when objects have been found to be unreachable, meaning that the mutators cannot read them.

The second part of the code in the implementation initializes the object and sets up its color. This atomic block is separate from the first because the collector, during tracing, may observe the uninitialized color of the newly allocated object.



(a) Bucket Data Structure

```
1. old <- Obj;
2. nw <- NextWrite[tid];        //@Local
3. Bucket[tid][nw] := old;      //@Local
4. Bucket[tid][nw +1] := val;   //@Local
5. NextWrite[tid] := nw +2;
6. fence;
7. Obj := val
```

(b) Low

```
1. old <- Obj;
   atomic {
2.   nw <- NextWrite[tid];
3.   Bucket[tid][nw] := old;
4.   Bucket[tid][nw +1] := val;
5.   NextWrite[tid] := nw +2
6. };
7. Obj := val
```

(c) High

Fig. 7: A concurrent write barrier applying MAKESTOREATOMIC, followed by three applications of GROWATOMICLOCAL.

4.2. Case Study: A Concurrent Write Barrier

Figure 7 shows the definition of a write barrier used to synchronize between mutators and the DLG collector. Because mutation can take place concurrently with a collection cycle, every write entails marking both the object currently pointed by the field being modified, as well as the newly stored object, with the color *Gray*. An object marked *Gray* indicates that it is potentially alive, and therefore not subject to collection.

Marking objects *Gray* is achieved by adding their pointers into a "per mutator" buffer of gray marked objects. This buffer is incremented by the mutator on one end, and visited by the collector thread on the other end. To enforce this protocol, depicted in Figure 7a, each mutator keeps a per-mutator global variable (`NextWrite[tid]`, only read/written by the owning mutator `tid`, and read by the collector). This variable indicates the writing end of the buffer. The actual insertion of an object to the mutator's buffer is *published* to the collector upon the increment of the `NextWrite[tid]` variable. Abstractly, it can be considered as performing three indivisible steps – a first reading step, in which the old object pointed by the field to be modified is fetched; a second marking step that marks the old object and the new object to be stored; and finally the writing step, in which the store is actually performed and made visible to the collector. The invariant being enforced is that marking happens in its entirety before the field update.

The program fragment in Figure 7b exposes two potential race conditions: the first can arise because of concurrent accesses to `Obj` by concurrently executing mutators; the second is due to concurrent accesses to the mark buffer by the mutator and the collector, that allows the collector to trace the new and old objects concurrently with the mutator performing the update. Both of these race conditions have been considered in

the design of the GC algorithm, and are benign. Hence our implementation does not enforce synchronization on these accesses. The Local annotations on the other hand make clear that for the refinement of this code it is assumed that `NextWrite[tid]` cannot be modified by any other thread (the environment). This is in fact true because this is a per-thread variable, only written by one thread, and read by that thread and the collector (cf. the `Flag` variable in Section 3). Similarly the publication of the bucket items through `NextWrite[tid]` guarantees that until the assignment to `NextWrite[tid]`, the bucket items at `Bucket[tid][NextWrite[tid]]` and `Bucket[tid][NextWrite[tid]+1]` will be inaccessible (either to read or write) to the collector until that assignment.

Our refinement technique proves that this program is equivalent to the specification shown in Figure 7c written using atomic blocks that clearly exposes the desired atomicity properties of the barrier obfuscated in the low-level representation of Figure 7b. Applying rules MAKESTOREATOMIC and GROWATOMICLOCAL three times, we get the atomic block shown in Figure 7c.

4.3. Results

Table I quantifies the benefits of our approach on salient components of the collector listed in the first column to give a sense of the complexity of the injected code snippets we consider before and after refinement. In the second and third columns, we count the number of atomic steps required for the snippet to terminate, and the number of loops that it contains. Many low level implementations are not guaranteed to terminate when subject to repeated interference (a livelock); thus potentially taking an "unbounded" number of steps. In our case livelocks are extremely unlikely and correspond to the possibility of threads not being scheduled, and therefore not cooperating. Whenever we cannot bound the number of steps used by a piece of code to terminate we use the symbol ∞. The idea is that under a fair scheduler these pieces of code terminate, except of course in the case of the lock implementation where there is a deadlock.

Component	(low, loop)	(high, loop)	Thds	CAS	Races	Rules	LoC
Lock	(∞,2)	(1,0)	all	✓	1	10	9
Alloc (Treiber)	(∞,1)	(2,0)	all	✓	heap size	21	15
Write Barrier	(7,0)	(3,0)	all	×	4	10	16
Cooperate	(#roots,2)	(2,0)	2	×	#roots + 2	18	25
Thread Spawn	(∞,2)	(2,0)	all	✓	#mutators	11	32
Thread Start	(∞,1)	(1,0)	2	×	1	7	5
Thread End	(∞,1)	(2,0)	all	✓	2	13	14
Func. Start	(#vars,1)	(2,0)	1	×	0	6	12
Func. End	(2,0)	(1,0)	1	×	0	6	3
Global Setup	(#obj,1)	(1,0)	1	×	0	6	20
Trace (GC)	(∞, 5)	(∞, 5)	all	✓	#thds + #obj	46	40
Sweep (GC)	(∞, 3)	(#obj, 3)	all	✓	#thds + #obj	18	29

Table I: Injected codes for a concurrent GC.

The code snippets in the table are injected into routines that typically involve memory allocation, and thus interact with the collector. For example, thread creation requires allocating and initializing private memory for each new thread. This piece of memory is safely acquired by first acquiring a lock, then scanning for a free block with a loop and upon success, releasing the lock. When the thread terminates, it releases its private memory with a similar procedure. A freshly spawned thread waits for the end of the collector's sweeping phase before starting. This wait loop is turned into a

Arrow	Synchronizes	Meaning
$\xrightarrow{ev}_{\nearrow}$		Single-thread Contribution
$\xrightarrow{ev}_{\searrow}$		TSO Memory Machine
$\xrightarrow{ev}_{t}$	$\xrightarrow{ev}_{\nearrow}$ $\xrightarrow{ev}_{\searrow}$	Memory and Threads Composition (no atomics)
$\rightarrow_{t}$	$\xrightarrow{ev}_{t}$ $\xrightarrow{ev}_{\nearrow}$	Full System Composition
$\overset{tr}{\hookrightarrow}$	$\xrightarrow{ev}_{\searrow}$ $\overset{tr}{\hookrightarrow}$	Abstract Environment Trace
$\overset{tr}{\Rightarrow}$	$\overset{tr}{\hookrightarrow}$ $\overset{tr'}{\Rightarrow}$	Single-thread with Abstract Environment

Fig. 8: Synchronization of the Different Semantics

single atomic block with an explicit assumption. The full write barrier, based on the code Figure 7, contains three atomic blocks that includes two exceptional cases. Since each mutator must keep track of its own roots, function prologues and epilogues are inserted to manage the representation of the root set. We note that the ostensible proof burden for the collector, when considering the number of steps and interleavings as the metric, is not greatly reduced using our refinement methodology; however, our approach does more closely align the steps of the high-level refined code with a semantics of a realistic relaxed-memory aware compiler backend bytecode, leading to an overall simplification in the final simulation argument.

The fourth column shows the degree of thread interference induced by the various components. The fifth column indicates whether the code contains a CAS instruction (generally used to implement synchronization). The sixth column shows the number of shared locations possibly involved in data races; for example, in the allocator, every cell of the free list is potentially subject to a data race. The final two columns count the number of tactics required to refine the code fragments and the lines of code in the low-level implementation, resp.

Notice that the size in LoC of the code snippets considered does not reflect the complexity involved their proof, and moreover, in the proofs of larger codes using them as a service. This is because these are the pieces of code that are highly concurrent, contain injected code, and contain data races. In other words, the pieces of code that do not stand the standard verification techniques used throughout the CompCertTSO framework. On the other hand, coarsening the atomicity of these pieces of code, liberates other parts of the proof from reasoning about complex interleavings possible due to these snippets. Much of the verification of the other components of a certified compiler can be carried out in the same way as it is done in Compcert and CompCertTSO, which we do not discuss in this paper.

THEOREM 4.1. *For every pair of code snippets* (s_l, s_h) *injected by our compiler* (*Table I included*)*, we have* $s_l \preccurlyeq s_h$. (COQPROOF)

Our formal development and compiler comprises roughly 20K lines of Coq, divided roughly in half between specifications and proofs. The full development can be found at http://www.irisa.fr/celtique/ext/inject/.

5. FORMALIZATION

5.1. Semantics

In this section, we present the semantics that justify our methodology. Figure 8 presents the different relations (arrows) we use, and the way in which they synchronize. We start our discussion with the semantics of $\mathcal{I}$. We elide the semantics of $\text{RTL}^{\mathcal{I}}$, which is simply the semantics of the RTL language of CompCertTSO with the addi-

tional commands of $\mathcal{I}$. As mentioned before, only terms in $\mathcal{I}_{\mathsf{L}}$, the low level commands of $\mathcal{I}$, are compiled into RTL. Terms in $\mathcal{I}_{\mathsf{H}}$ need not have an obvious implementation in RTL, and only serve to facilitate our proofs.

LANGUAGE : $\mathcal{I}$

$$\begin{array}{rcl} s \in \mathcal{I} & ::= & \mathsf{skip} \mid d = op(\vec{r}) \mid s; s \mid \mathsf{if}\ c\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{repeat}\ s\ \mathsf{until}\ c \\ & \mid & \{\pi\}\,\mathsf{load}_{\nu}(d,r) \mid \{\pi\}\,\mathsf{store}_{\nu}(d,r) \mid \mathsf{cas}(d,r,o,n) \mid \mathsf{fence} \mid \mathsf{abort} \\ & \mid & \mathsf{atomic}\ s \mid \mathsf{assume}\ c \mid \mathsf{branch}\ s,\ s \mid \mathsf{loop}\ s \\ \nu & ::= & \mathsf{Local} \mid \epsilon \end{array}$$

EVENTS : $\mathcal{M}Ev$, Ev

$$\begin{array}{rcl} e \in \mathcal{M}Ev & ::= & \mathsf{rd}_{p,v} \mid \mathsf{st}_{p,v} \mid \mathsf{cas}_{p,v,v',w} \mid \mathsf{ubff}_{p,v} \mid \# \\ ev \in Ev & ::= & e \mid \tau \mid \triangleright \mid \triangleleft \mid \dagger \end{array}$$

STEP EVALUATION : $(\mathcal{I} \times \mathit{RegMap}) \xrightarrow{ev} (\mathcal{I} \times \mathit{RegMap})$

$$\begin{array}{rcll} \mathsf{load}_{\nu}(d,r),\ rs[r:p] & \xrightarrow{\mathsf{rd}_{p,v}} & \mathsf{skip},\ rs[d \leftarrow v] & \\ \mathsf{store}_{\nu}(d,r),\ rs[d:p, r:v] & \xrightarrow{\mathsf{st}_{p,v}} & \mathsf{skip},\ rs & \\ \mathsf{cas}(d,r,o,n),\ rs[r:p,\ o:v, n:v'] & \xrightarrow{\mathsf{cas}_{p,v,v',w}} & \mathsf{skip},\ rs[d \leftarrow w] & \\ d = op(\vec{r}),\ rs & \longrightarrow & \mathsf{skip},\ rs[d \leftarrow \mathcal{O}(op,\vec{r})] & \\ \mathsf{skip}; s,\ rs & \longrightarrow & s,\ rs & \\ \mathsf{if}\ c\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2,\ rs & \longrightarrow & s_1,\ rs & \mathsf{if}\ \mathcal{C}(c,rs) \\ \mathsf{if}\ c\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2,\ rs & \longrightarrow & s_2,\ rs & \mathsf{if}\ \neg\mathcal{C}(c,rs) \\ \mathsf{repeat}\ s\ \mathsf{until}\ c,\ rs & \longrightarrow & \left(\begin{array}{l} s; \mathsf{if}\ c\ \mathsf{then} \\ \quad \mathsf{repeat}\ s\ \mathsf{until}\ c \\ \mathsf{else\ skip} \end{array}\right),\ rs & \\ \mathsf{fence},\ rs & \xrightarrow{\#} & \mathsf{skip},\ rs & \\ \mathsf{loop}\ s,\ rs & \longrightarrow & (s; \mathsf{loop}\ s),\ rs & \\ \mathsf{loop}\ s,\ rs & \longrightarrow & \mathsf{skip},\ rs & \\ \mathsf{branch}\ s_1,\ s_2,\ rs & \longrightarrow & s_i,\ rs & i \in \{1,2\} \\ \mathsf{assume}\ c\,,\ rs & \longrightarrow & \mathsf{skip},\ rs & \mathsf{if}\ \mathcal{C}(c,rs) \\ \mathsf{atomic}\ s,\ rs & \xrightarrow{\triangleright} & s; \mathtt{endatomic},\ rs & \\ \mathtt{endatomic},\ rs & \xrightarrow{\triangleleft} & \mathsf{skip},\ rs & \\ \mathsf{abort},\ rs & \xrightarrow{\dagger} & \mathsf{skip},\ rs & \end{array}$$

$$\frac{s_0,\ rs \xrightarrow{ev} s_0',\ rs'}{(s_0; s_1),\ rs \xrightarrow{ev} (s_0'; s_1),\ rs'} \qquad \frac{s_0,\ rs \xrightarrow{\dagger} s_0',\ rs'}{(s_0; s_1),\ rs \xrightarrow{\dagger} \mathsf{skip},\ rs'}$$

Fig. 9: Events and thread-local semantics of $\mathcal{I}$.

Our semantics are structured as the composition of different labeled transition systems. Figure 9 repeats the language of Figure 3 for reference and presents the events and small-step semantics of individual commands of the $\mathcal{I}$ language. Notice that we have added placeholders $\{\pi\}$, standing for *assertion predicates*, to the syntax of load and store instructions. These predicates will not be used in the definition of the program semantics but are necessary to support the rely/guarantee proof methodology we introduce later. Hence, we postpone their treatment and omit them from the presentation of the semantics.

Our labels are composed of memory, synchronization, and error events. Memory events $\mathcal{M}Ev$, roughly correspond to the memory operations available in the x86 architecture. These include: reads $\mathsf{rd}_{p,v}$, representing the query of memory location p which returns value v; writes $\mathsf{st}_{p,v}$, representing the result of a store to a location of a value v found in a memory location p; compare-and-set events $\mathsf{cas}_{p,v,v',w}$, representing an atomic read-modify operation on memory location p where v is the expected value, v' is the value to be stored in p and w is the result of the read – notice that the update is executed only if v and w coincide; an event recording the execution of a memory fence $\#$; and a special event to denote the flush of a TSO buffer $\mathsf{ubff}_{p,v}$. The full set of events Ev includes memory events as well as a τ (empty event) corresponding to a thread-local operation; we omit such labels in general; $\triangleright$ and $\triangleleft$ events, representing the beginning and the end of an atomic command respectively; and an abort event, $\dagger$, generated by the abort command to represent exceptional execution.

The semantics of Figure 9 represents the contribution of each thread, through events, to the overall system. Figure 10 shows the the small-step semantics of the composition of different threads and their interaction through shared-memory. Recall that based on the syntax of Figure 3, metavariables $r, o, n, d \in \textit{Registers}$ represent registers, v ranges over values, and p represents a memory location. We distinguish the sublanguage $\mathcal{I}_{\mathsf{L}}$ of $\mathcal{I}$ by disallowing the high-level statements for $\mathcal{I}$ (i.e., assume, loop, branch and atomic).

Thread local evaluation is defined by a small-step evaluation judgment of the form $s,\ rs \xrightarrow{ev}_{\tau} s', rs'$, where s and s' are commands in $\mathcal{I}$, $rs, rs' \in \textit{RegMap}$ represent register maps, associating values to the registers of the thread. We use the command skip to represent termination. The notation $rs[r : v]$ denotes a register map that associates v to the register r. The judgment states that evaluating statement s with a register map rs yields a state with continuation s' and a new register map rs' while emitting the event ev. Notice that when an abort command is executed, the whole command is immediately terminated – with continuation skip and abort event $\dagger$. Since load and compare-and-set judgments are defined in isolation from the memory judgments, but depend on the memory, their rules are non-deterministic. For example a $\mathsf{rd}_{p,v}$ step must admit every possible value v as its return value. The value is only constrained when synchronizing with the memory, where only one value can be read. The property of accepting all possible return values is called *receptiveness* in CompCertTSO [Ševčík et al. 2013], and our semantics uses the same principle.

Statements fence and $\mathsf{cas}(d, r, o, n)$ emit the events $\#$ and $\mathsf{cas}_{p,v,v',w}$ respectively with the obvious semantic rules. For the latter instruction, the memory location to be read-and-modified is contained in the register r. Hence, if the register r contains a pointer p; the expected value for the pointer, given in register o, is v; the value to write, in register n, is v'; and the actual value of p in memory is w, the instruction generates the event: $\mathsf{cas}_{p,v,v',w}$. The value w is placed in the destination register d. In the case where $v = w$, the location p is updated to v', otherwise it remains unchanged. Here also, the rule for $\mathsf{cas}_{p,v,v',w}$ is *receptive*. Rules related to local control flow emit τ events, whose labels we omit since their effect is not observable for other threads.

The command loop s nondeterministically chooses to either execute the statement s and continue looping, or terminate immediately. The statement branch $s_1,\ s_2$ nondeterministically chooses to execute s_1 or s_2. The command assume c only proceeds if register map rs satisfies the condition c. The atomic s command executes s atomically, ensuring that the effect of the atomic action is propagated to memory from the local store buffer upon completion; `endatomic` is a *runtime statement* simply used as a marker to record the end of an atomic section. It is not part of the source code syntax. Finally, we use

MEMORY : $\mathit{Mem} \xrightarrow{ev}_{\mathit{Tid}} \mathit{Mem}$ WITH $\mathit{Mem} = ((\mathit{Refs} \rightharpoonup \mathit{Value}) \times (\mathit{Tid} \rightharpoonup \mathit{Buf}))$

$$\frac{v = \mathsf{lastIn}(M.\mathsf{b}(t),\, p)_{M.\mathsf{m}\,(p)}}{M \xrightarrow{\mathsf{rd}_{p,v}}_t M} \qquad \frac{M' = \mathsf{bufferPush}(M, t, \langle p, v\rangle)}{M \xrightarrow{\mathsf{st}_{p,v}}_t M'}$$

$$\frac{\mathsf{CAS}\, M\, p\, v\, v' \;=\; (w, M') \qquad M.\mathsf{b}(t) = \mathsf{emptyBuff}}{M \xrightarrow{\mathsf{cas}_{p,v,v',w}}_t M'} \qquad \frac{M.\mathsf{b}(t) = \mathsf{emptyBuff}}{M \xrightarrow{\#}_t M}$$

$$\frac{\mathsf{bufferPop}(M, t) = (\langle p, v\rangle, M') \neq \bot \qquad M'' = \mathsf{updateMem}(M',\, \langle p, v\rangle)}{M \xrightarrow{\mathsf{ubff}_{p,v}}_t M''}$$

MEMORY COMPOSITION : $(\mathit{Mem} \times \mathit{ThrdSt}) \rightarrow_{\mathit{Tid}} (\mathit{Mem} \times \mathit{ThrdSt})$

MEMORY STEP
$$\frac{st \xrightarrow{ev} st' \qquad M \xrightarrow{ev}_t M'}{(M, st) \xrightarrow{ev}_t (M', st')}$$

INTRA STEP
$$\frac{st \rightharpoonup st'}{(M, st) \rightarrow_t (M, st')}$$

UNBUFFER
$$\frac{M \xrightarrow{\mathsf{ubff}_{p,v}}_t M'}{(M, st) \xrightarrow{\mathsf{ubff}_{p,v}}_t (M', st)}$$

THREAD COMPOSITION : $(\mathit{Mem} \times \mathit{ThrdMap}) \rightarrow_{\mathit{Tid}} (\mathit{Mem} \times \mathit{ThrdMap})$

INTERLEAVE NONATOMIC
$$\frac{(M, \Pi(t)) \xrightarrow{ev}_t (M', st') \qquad ev \neq \triangleright}{(M, \Pi) \rightarrow_t (M', \Pi[t \leftarrow st'])}$$

INTERLEAVE ATOMIC
$$\frac{\Pi(t) \xrightarrow{\triangleright} st' \qquad (M, \Pi[t \leftarrow st']) \rightarrow_t^* (M', \Pi') \qquad M'.\mathsf{b}(t) = \mathsf{emptyBuff} \qquad \Pi'(t) \xrightarrow{\triangleleft} st''}{(M, \Pi) \rightarrow_t (M', \Pi'[t \leftarrow st''])}$$

Fig. 10: Memory and Thread Composition Semantics

the command skip to denote termination, a thread whose only command is skip is considered terminated.

In Figure 10 we present the semantics of thread composition stratified into two parts: (1) the semantics of the memory machine, and (2) the overall system behavior composing the memory and the threads. A thread system whose threads are all terminated is considered a terminated program.

The memory machine implements the TSO memory model following the guidelines of CompCertTSO. The memory state, which shall remain abstract throughout the paper, contains a *store*, mapping memory locations to values, and a *write buffer* for each thread. A write buffer is simply a FIFO queue of store events of the form $\langle p, v\rangle$ (a pending store of a value v at address p). Given a memory $M \in \mathit{Mem}$, we use projections $M.\mathsf{m}$ and $M.\mathsf{b}$ to obtain the store and the buffer map, resp.; $M.\mathsf{b}(t)$ represents the buffer of thread t and the operations $\mathsf{bufferPush}(M, t, \langle p, v\rangle)$, $\mathsf{bufferPop}(M, t)$, $\mathsf{updateMem}(M,\, \langle p, v\rangle)$ and $\mathsf{emptyBuff}$ have the obvious meanings, where M is a memory state, t a thread. $\mathsf{lastIn}(B,\, p)$ returns the value of the last-in item in the store buffer B for the location p. Finally, the operation $\mathsf{CAS}\, M\, p\, v\, v' \;=\; (w, M')$ returns the pair containing the value w read in the memory M for pointer p, and accordingly the new memory M' (which will differ from M in case the operation was successful.)

The semantics of the memory machine is described by judgments of the type $M \xrightarrow{ev}_t M'$ which represent the execution of an event ev by thread t and that modifying the

memory state M into the state M'. These rules closely follow the memory machines described in [Ševčík et al. 2013; Burckhardt et al. 2012]; note that in the rule for reading, we use the notation $\mathsf{lastIn}(M.\mathsf{b}(t),\, p)_{M.\mathsf{m}\,(p)}$ to indicate that the absence of location p in the store buffer for thread t – i.e., $p \notin \mathrm{dom}(M.\mathsf{b}(t))$ – results in reading the contents of p from memory: $M.\mathsf{m}\,(p)$. Note that the unbuffering is the only memory operation that is not derived from the program syntax; it can be applied at any time when the thread buffer is non-empty, and flushes some unspecified portion of the buffer to memory.

The state of the whole system is comprised of two components, a global memory, and a thread map (Π), which maps thread identifiers to thread states; these states contain the registers and the code of the thread. There are two judgments in this semantics. Judgments of the form: $(M, st) \xrightarrow{ev}_t (M', st')$, where st is the thread state for thread t, containing t's continuation and register map. The judgment represents the execution of a step by t with respect to shared memory M. The rule MEMORY STEP synchronizes the semantics of individual threads and the memory system by having the events in the premises and in the consequent coincide. The rule INTRA STEP does not need to exercise the memory machine, and the rule UNBUFFER asynchronously flushes elements of the buffer into the memory without modifying the thread state.

Thread composition judgments have the shape $(M, \Pi) \rightarrow_t (M', \Pi')$. This semantics captures in a single step, the multiple steps that could be required to execute an atomic statement. The rule INTERLEAVE NONATOMIC executes any statement labelled with an event other than $\triangleright$. The rule INTERLEAVE ATOMIC executes the atomic statement in a single step thereby ensuring that all actions in the atomic statement occur without interleaving of other threads – observe that the thread identifier in the premise restricts the multistep in the premise to only execute steps of thread t.

5.2. Soundness

The starting point of our refinement methodology assumes that input programs are verified using a rely/guarantee-style program logic [Jones 1983; Feng 2009; Liang et al. 2012a], a well-established technique for verifying shared-memory concurrent programs. As such, we assume that programs in $\mathcal{I}_{\mathsf{L}}$ are annotated with assertions and rely/guarantee conditions supporting a rely/guarantee proof. Since we are not concerned with the actual proof here, we adopt a shallow embedding of rely/guarantee conditions and assertions which we use to enhance our refinements. In this section, we briefly define the notions of annotations and rely/guarantee conditions, and give an overview of the rely/guarantee proof structure.

Assertions π that accompany certain instructions in our code have the following signature:

$$\pi \in \mathcal{P}re : \textit{Mem} \times \textit{RegMap} \rightarrow \mathsf{bool}$$

The similarity with Hoare-logic [Hoare 1969] style annotations should be immediate. The meaning of these annotations is that for any state that reaches an instruction annotated with such an assertion, evaluating the assertion at that state should return true. The following definition formalizes this intuition.

Definition 5.1 (*Correct Assertions*). A RTL$^{\mathcal{I}}$ program R satisfies its attached assertions if:

$$\forall\, t, \pi, rs, s,\ \forall (M, \Pi) \in \mathsf{reach}^{\mathcal{I}}(\mathsf{R}),\ \Pi(t) = (s, rs) \wedge \pi \mathbin{@} s \Rightarrow \pi(M, rs)$$

where $\pi \mathbin{@} s$ is the projection of the assertion π from the first command of s defined by:

$$\frac{}{\pi \mathbin{@} \{\pi\}\, \mathsf{load}_\nu(d, r)} \qquad \frac{}{\pi \mathbin{@} \{\pi\}\, \mathsf{store}_\nu(d, r)} \qquad \frac{\pi \mathbin{@} s_1}{\pi \mathbin{@} s_1; s_2}$$

and $\mathsf{reach}^{\mathcal{I}}(\mathsf{R})$ is the set of states reachable from the input of the global program R, where we only consider the state of threads currently engaged in injected code – that is, whose continuation is a statement $s \in \mathcal{I}$ rather than RTL.

Additionally, rely/guarantee adds *rely* and *guarantee* conditions. These conditions allow us to reason about concurrency in a thread-modular way. Rely conditions represent a thread's expectations of the state changes made by the environment (i.e., other threads). Guarantee conditions represent the state changes made by a thread to the environment.

Not all actions in a rely are allowed at any state, and therefore a rely condition takes a thread identifier t as a parameter, a memory M, and returns a set containing the "actions" that the environment of this thread can do starting from M. This choice of rely guarantee logic is similar to [Ridge 2010]. In our shallow embedding of rely/guarantee, we abstract memory actions with the events of Figure 9. A rely condition, ranged over by the meta-variable $\rho \in \mathcal{R}ely$, has the following signature:

$$\rho[t] \in \mathcal{R}ely : \boldsymbol{Mem} \rightarrow 2^{\mathcal{M}Ev}$$

where $\mathcal{R}ely$ is the domain of rely conditions. In fact, guarantee conditions, which we range with the metavariable θ have the exact same signature, but their interpretation is different. We write $\theta[t]$ for the guarantee of thread t.

The usage of rely/guarantee conditions is best explained using an idealized presentation of some of the standard proof rules. Consider for example a generic proof rule for the store command.

$$\frac{\mathsf{stable}(P, \rho[t]) \quad \{P\}\ \mathsf{store}(d, r)\ \{Q\} \quad \mathsf{stable}(Q, \rho[t]) \quad [\![\mathsf{store}(d, r)]\!](P) \subseteq \theta[t]}{\rho, \theta \vdash_t \{P\}\ \mathsf{store}(d, r)\ \{Q\}}$$

where we denote by: $\mathsf{stable}(P, \rho[t])$ the condition that the assertion P must remain valid after applying to any memory that satisfies P initially, any of the events in $\rho[t]$. In rely/guarantee jargon, P is closed under the rely $\rho[t]$ if (a) $\{P\}\ \mathsf{store}(d, r)\ \{Q\}$ holds - this is the usual Hoare-triple proof in sequential Hoare-logics; and (b) if $[\![\mathsf{store}(d, r)]\!](P) \subseteq \theta[t]$ - the notation asserts that any memory event that could be generated by the command $\mathsf{store}(d, r)$ starting in a memory satisfying P must be permissible according to the guarantee condition $\theta[t]$ of thread t (see [Ridge 2010] for a similar argument). Summarizing, this proof rule requires that the assumptions made in P and Q should be valid under the assumed interference given by $\rho[t]$, and that the actions made by the command $\mathsf{store}(d, r)$ must be allowed by the guarantee $\theta[t]$.

Of course a rely/guarantee proof would not be sensible if the assumptions made about the environment by each thread (in $\rho[t]$) are not met by the other threads. That is exactly the purpose of the guarantee ($\theta[t]$) in each thread. The final proof obligation requires that for each thread t we verify that any other threads t' guarantee has been considered as possible interference; we then have that

$$\forall t, t',\ t \neq t' \Rightarrow \theta[t'] \subseteq \rho[t]$$

This is the top level proof obligation of a rely/guarantee proof.

In this work we assume the correctness of rely guarantee conditions, and establish this assumption semantically using the following definition.

Definition 5.2 (*Correct Rely Condition*). A $\mathrm{RTL}^{\mathcal{I}}$ program R satisfies the rely conditions ρ if: $\forall\ t, t', (M, \Pi) \in \mathsf{reach}^{\mathcal{I}}(\mathsf{R})$,

$$(M, \Pi) \xrightarrow{ev}_{t'} (M', \Pi')\ \wedge t \neq t' \Rightarrow ev \in \rho[t](M)$$

$$\frac{}{\rho \vdash_t M \overset{\epsilon}{\hookrightarrow} M} \qquad \frac{M \xrightarrow{e}_{t'} M' \qquad e \in \rho[t](M) \qquad t' \neq t \qquad \rho \vdash_t M' \overset{tr}{\hookrightarrow} M''}{\rho \vdash_t M \overset{e::tr}{\hookrightarrow} M''}$$

ATOMIC
$$\frac{\big(M, \{t \mapsto (s, rs)\}\big) \rightarrow_t \big(M', \{t \mapsto (\mathsf{skip}, rs')\}\big)}{\rho \vdash_t s : (M, rs) \overset{\varepsilon}{\Rightarrow} (M', rs')}$$

SEQUENCE
$$\frac{\rho \vdash_t s_1 : (M, rs) \overset{tr_1}{\Longrightarrow} (M', rs') \quad \rho \vdash_t s_2 : (M', rs') \overset{tr_2}{\Longrightarrow} (M'', rs'')}{\rho \vdash_t s_1; s_2 : (M, rs) \overset{tr_1 \cdot tr_2}{\Longrightarrow} (M'', rs'')}$$

UNBUFFER
$$\frac{M \xrightarrow{\mathsf{ubff}_{p,v}}_t M_1 \quad \rho \vdash_t s : (M_1, rs) \overset{tr}{\Rightarrow} (M_2, rs') \quad M_2 \xrightarrow{\mathsf{ubff}_{p',v'}}_t M'}{\rho \vdash_t s : (M, rs) \overset{tr}{\Rightarrow} (M', rs')}$$

COMPOSE
$$\frac{\rho \vdash_t M \overset{tr_1}{\hookrightarrow} M_0 \quad \rho \vdash_t s : (M_0, rs) \overset{tr_2}{\Rightarrow} (M_1, rs') \quad \rho \vdash_t M_1 \overset{tr_3}{\hookrightarrow} M'}{\rho \vdash_t s : (M, rs) \overset{tr_1 \cdot tr_2 \cdot tr_3}{\Longrightarrow} (M', rs')}$$

Fig. 11: Semantics of Expressions – Abstract Environment Respecting the Rely (Excerpt)

This means that from any reachable state (M, Π), any step performed by a thread other than t, must respect the rely condition of t.

5.3. The Refinement Relation

Because the definition of refinement has to take into account the behavior of the environment, we develop a semantic interpretation of the refinement relation that relies on a dedicated semantic judgment of the $\mathcal{I}$ language and is parametric on a rely predicate ρ provided by the programmer (or prover) again as part of an underlying rely/guarantee proof. The only purpose of this semantics is to define and enable the refinement methodology, and it has no computational effect. This semantics represents a single thread's view, executing with an arbitrary context or *environment* (cf. [Brookes 1993]) respecting the rely condition ρ, which in this work is assumed to be given.

Figure 11 provides the inductive rules of the judgment $\rho \vdash_t s : (M, rs) \overset{tr}{\Rightarrow} (M', rs')$ indicating that starting from a memory M and register map rs the thread t can execute the statement s terminating in a state (M', rs') in a context where the *environment*, an over-approximation of the other threads running in parallel, modify the store by performing the memory events contained in the sequence tr. Moreover, these events are checked to satisfy the rely assumptions in ρ. In the following, tr is a sequence of read, store, and unbuffering events that capture accesses and modifications to the store made by other threads, and the memory M, for the executing thread (in this case t). This semantic interpretation allows us to focus on the evaluation steps taken by one thread, merging the interleavings of other threads into an external environment captured by tr.

The rules for environment steps are presented in the first two rules of Figure 11, defining the semantic transition $\overset{tr}{\hookrightarrow}$. Notice that the environment is only allowed to

modify the memory by respecting the ρ condition, in which $\rho[t](M)$ returns a set of events that can be performed by the environment. In the upper rules, we verify that the event the environment attempts (either a $\mathsf{st}_{p,v}$, $\mathsf{ubff}_{p,v}$ or a $\mathsf{rd}_{p,v}$) is not disallowed by the rely. The remaining rules, of which we only present the most significant ones, are unsurprising and deal with steps taken by the command s, the composition of environment traces with the current command, the unbuffering of TSO buffers, and control-flow. They largely resemble the rules of Figure 10.

Example. Assuming a universal rely (i.e., a rely condition that permits the environment to generate any event), the statement:

```
x := 0; x := 1; x := 2
```

may be interleaved with the following external trace of events:

$$\langle \mathsf{rd}_{\mathtt{x},0}, \mathsf{rd}_{\mathtt{x},1}, \mathsf{rd}_{\mathtt{x},2} \rangle$$

This trace captures the fact that other threads in the environment may witness three updates to `x` performed by the executing thread. However, this trace cannot be produced by executing:

```
x := 0; atomic{x := 1; x := 2}
```

The atomic action prevents interleaving of other threads and thus the intermediate update of 1 to `x` cannot be observed.

From Rely/Guarantee Conditions to Syntactic Annotations. Our refinement methodology leverages automation through purely syntactic rules. However, our proofs only provide us with rely/guarantee conditions to express invariants about environments. We show how these invariants can be used to justify the use of our syntactic refinement rules such as GROWATOMICLOCAL (Figure 5).

Consider the Local annotations that we attached to the code in some memory accessing operations. These indicate to our refinement rules that the memory locations being manipulated by the executing thread cannot be modified by other threads. These annotations, which we used in Section 2, take the form: $\mathsf{load}_{\mathsf{Local}}(d, r)$ and $\mathsf{store}_{\mathsf{Local}}(d, r)$. These annotations encode in the program syntax, facts that are assumed to be true from the rely conditions that accompany the code. We can therefore define what it means for an annotation to be correct.

Definition 5.3 (*Correct Local Annotations*). Given a statement s such that $s \in \{\{\pi\}\, \mathsf{load}_{\mathsf{Local}}(d, r), \{\pi\}\, \mathsf{store}_{\mathsf{Local}}(d, r)\}$ and s appears in an injected code block of thread t, we say that s has a correct local annotation according to ρ if the two following conditions hold:

if $s = \{\pi\}\, \mathsf{load}_{\mathsf{Local}}(d, r)$,

$$\forall M, rs,\ \pi(M, rs) \Rightarrow \{\mathsf{st}_{rs(r),\_}, \mathsf{ubff}_{rs(r),\_}, \mathsf{cas}_{rs(r),\_,\_,\_}\} \cap \rho[t](M) = \emptyset$$

if $s = \{\pi\}\, \mathsf{store}_{\mathsf{Local}}(d, r)$,

$$\forall M, rs,\ \pi(M, rs) \Rightarrow \{\mathsf{rd}_{rs(d),\_}, \mathsf{st}_{rs(d),\_}, \mathsf{ubff}_{rs(d),\_}, \mathsf{cas}_{rs(d),\_,\_,\_}\} \cap \rho[t](M) = \emptyset$$

In essence, we say that a Local annotation is correct if the rely condition ρ that accompanies the code s_l is sufficient to guarantee that other threads are not able to observe intermediate states of the execution of this piece of code through these variables.

Proving that the Local annotations added for refinement are correct is a proof obligation that we must discharge. Importantly, this proof obligation is only a logical consequence of the rely condition and the program assertions.

Example 5.4. Figure 12 presents the garbage collector code that traverses the mutators buckets (cf. the write barrier protocol presented in Figure 7b) and marks the

```
1.  {π1} nr <- NextRead[tid];      // @Local
2.  fence;
3.  nw <- NextWrite[tid];
4.  while nr < nw do
5.     {π2} b <- Bucket[tid][nr]; // @Local
6.     nr = nr +1;
7.     ... change the color of b (thread-local) ...
8.  od
9.  {π3} NextRead[tid] := nr;      // @Local
```

$$\pi_1 \equiv \lambda M\ rs.\ \mathsf{True} \equiv \pi_3$$

$$\pi_2 \equiv \lambda M\ rs.\ rs(\mathtt{nr}) < rs(\mathtt{nw}) \leq [\![\&\mathtt{NextWrite}[t]]\!](M)$$

$$\begin{aligned}
\rho(t_{\mathrm{GC}}) \equiv \lambda M.\ & \{\ \mathsf{st}_{p,v}, \mathsf{ubff}_{p,v}\ \mid \\
& \big(\forall t,\ p \neq [\![\&\mathtt{NextRead}[t]]\!](M)\big) && (1)\\
& \wedge \\
& \big(\forall t,\ p = [\![\&\mathtt{NextWrite}[t]]\!](M) \Rightarrow v > [\![\mathtt{NextWrite}[t]]\!](M)\big) && (2)\\
& \wedge \\
& \big(\forall t, i,\ p = [\![\&\mathtt{Bucket}[t][i]]\!](M) \Rightarrow i \geq [\![\mathtt{NextWrite}[t]]\!](M)\big)\ \} && (3)\\
\cup & \\
& \{\ \mathsf{rd}_{p,_}\ \mid\ \big(\forall t, p \neq [\![\&\mathtt{NextRead}[t]]\!](M)\big)\ \} && (4)
\end{aligned}$$

Fig. 12: GC-side Bucket Protocol: Tracing the Roots

white objects found as black, representing *alive* objects. In this code we present an inner loop that traverses the bucket of thread tid, and is executed when the GC is tracing the object graph. To that end, the GC thread holds in a GC-local table `NextRead`, indexed by the thread id of the mutators, the last seen element of the bucket of that thread. Recall that mutators indicate through their `NextWrite[tid]` variable the first unused bucket slot. This protocol ensures that the mutator can write any bucket element larger or equal than `NextWrite[tid]`, and that it will not modify, nor read, any object lower than that value. The GC is free to traverse the bucket up to `NextWrite[tid]` for each `tid`. Hence, `NextWrite[tid]` implements racy synchronization between the GC and the mutator `tid` to protect accesses to `Bucket[tid]`.

Let us now illustrate the assertions, rely conditions, and local annotation proof obligations that make this piece of code sound. We give the preconditions π_1, π_2, and π_3 in Figure 12 as well as the part of the rely guarantee ρ concerning the GC thread, denoted by t_{GC} relevant to this piece of code. Preconditions π_1 and π_3 are vacuous. Assertion π_2 requires that before the read of a bucket element by the GC in the bucket of thread `tid`, the registers `nr`, representing the next read of the collector, `nw`, representing the last seen next write value, and the actual value in the next write pointer `NextWrite[tid]` satisfy the ordering constraints of the bucket protocol.

The rely $\rho(t_{GC})$ allows any store, unbuffering and read event other than the ones restricted in (1), (2), (3) and (4) above. Of course, here we are showing only the rely conditions relevant to this piece of code, more constraints can limit stores and unbufferings on other portions of the memory, but we omit them here for didactic reasons. Stated in English: (1) indicates that the memory location found at $[\![\&\mathtt{NextRead}[\mathtt{t}]]\!](M)$

cannot be modified by any other thread.[3] This corresponds to the intuition that the `NextRead` table is thread-local to the GC. Condition (4) expresses the same intuition but limiting reads by other threads instead of just stores and unbufferings. Condition (2) states that, while the value of `NextWrite[t]` can change, it must do so in a monotonic way. We use this condition to ensure that the precondition π_2 is stable with respect to the rely conditions. Finally, condition (3) asserts that if a bucket element of thread t changes (in `Bucket[t]`), its index i must be larger than or equal to the current value of `NextWrite[tid]`.

With these conditions at hand, we can prove that the Local annotations found in the code are correct. The rely conditions (1) and (4) trivially imply the correctness of the annotations at lines 1 and 9. It remains to show that the annotation at line 5 is correct. It then suffices to observe that from the precondition π_2, `nr` is lower than the current value of `NextWrite[t]`; through the rely condition (3), it can be seen that no other thread, including t, can modify this bucket element, and therefore the proof obligation of Definition 5.3 is immediately met. A similar argument can be made in the code of Figure 7b.

We clarify that the proofs justifying the rely conditions as in the example above, have not been fully carried out in Coq, and are not considered a contribution of this paper. The methodology we present in this paper takes these rely conditions as a premise, but the rely/guarantee proof itself is not a novelty of our approach.

We can finally provide a definition conjoining the correctness of the assertions, the rely conditions and the Local annotations as follows.

Definition 5.5 (*Correctly Annotated Program*). An $\mathtt{RTL}^{\mathcal{I}}$ program R is correctly annotated with respect to a rely condition ρ if: (1) it satisfies its attached assertions (Definition 5.1), (2) it satisfies the rely condition ρ (Definition 5.2), and (3) every local load/store that appears in it has a correct local annotation according to ρ (Definition 5.3).

We denote this fact by $[\![\mathsf{R}]\!](\rho)\checkmark$. We overload the notation so that $[\![s]\!](\rho)\checkmark$ denotes an injected statement s that appears in a program R satisfies $[\![\mathsf{R}]\!](\rho)\checkmark$.

With this we can give our full definition of refinement.

Definition 5.6. Let $s_h \in \mathcal{I}$, $s_l \in \mathcal{I}_{\mathsf{L}}$ and $\rho \in \mathit{Tid} \to \mathcal{R}ely$. We say that s_l refines s_h under the rely condition ρ (denoted $\rho \vDash s_l \preccurlyeq s_h$), if for all states (M, rs), (M', rs') and all traces tr, and all t we have that

$$\rho \vdash_t s_l : (M, rs) \overset{tr}{\Rightarrow} (M', rs') \text{ implies } \rho \vdash_t s_h : (M, rs) \overset{tr}{\Rightarrow} (M', rs')$$

Thus, if starting in state (M, rs) which satisfies the precondition of s_l, threads other than t are able to perform a trace of memory actions tr during t's execution of s_l, which terminates in state (M', rs'), there must exist an execution of s_h from (M, rs) that also terminates in (M', rs') and permits the same trace tr of environment memory actions. In both cases the trace tr must respect the rely condition imposed by ρ. Notice that this semantic notion of refinement needs only consider the code of one thread at a time. The abstract notion of environment quantifies all possible threads that could be composed with this code. All we require is that the same environments be allowed for both pieces of code, under the assumptions made by ρ. In following statements we will additionally require non-failing terminating traces, to guarantee that semantics

[3] We use the notation $[\![\&\mathtt{NextRead[t]}]\!](M)$ as a shortcut to denote the memory address of the array element in memory M.

of the overall thread composition is preserved by refinement, but this is not necessary to prove the correctness of the rules.

We observe that

$$\vDash \texttt{x := 0; x := 1; x := 2} \not\preccurlyeq \texttt{x := 0; atomic\{x := 1; x := 2\}}$$

whereas,

$$\vDash \texttt{atomic \{x := 1; x := 2\}} \preccurlyeq \texttt{atomic \{x := 2\}}.$$

THEOREM 5.7. *If $s_h \in \mathcal{I}$, $s_l \in \mathcal{I}$ and ρ is a rely condition such that $[\![s_l]\!](\rho)\checkmark$ holds, then*

$$s_l \preccurlyeq s_h \Rightarrow \rho \vDash s_l \preccurlyeq s_h$$

where $s_l \preccurlyeq s_h$ is derived from the rules of Figure 5.

(COQ PROOF)

5.4. Embedding the Methodology in a Compiler

As explained earlier, our overall goal is to construct a whole-system backward simulation from target to source; this whole-program simulation is built from a series of smaller simulation steps that relate adjacent IRs in the compilation stack. We use the refinement methodology to prove an inclusion property between $\mathcal{I}_\mathsf{L}$ and $\mathcal{I}$. Recall that the $\mathcal{I}$ language is embedded within the $\text{RTL}^{\mathcal{I}}$ IR that encapsulates $\mathcal{I}$ terms in some of its nodes. This embedding enables a modular correctness proof, allowing us to reason about refinements on just the $\mathcal{I}$ sublanguage. For example, in Figure 1c statement (1) is an $\mathcal{I}$ statement, as is statement (1) in Figure 1b; both of these statements are embedded within what is otherwise an ordinary RTL program.

Because of this embedding, it is convenient to consider RTL contexts, $\mathsf{R}[]$, an RTL where one of its nodes contains a hole that can be filled with an $\mathcal{I}$ command s. A complete $\text{RTL}^{\mathcal{I}}$ program is simply an RTL context with no empty holes. With this simple notion of context, we can state our final soundness result. An RTL context may contain several holes, and in that case substitution is done starting from the leftmost hole in the context.

THEOREM 5.8 (SIMULATION FROM REFINES). *Consider a low statement $s_l \in \mathcal{I}_\mathsf{L}$ and an $\text{RTL}^{\mathcal{I}}$ program $\mathsf{R}[s_l]$ such that $\mathsf{R}[s_l]$ is correctly annotated by a rely condition ρ ($[\![\mathsf{R}[s_l]]\!](\rho)\checkmark$). Moreover, consider a high statement $s_h \in \mathcal{I}$ such that $s_l \preccurlyeq s_h$. Then, for each non-failing terminating trace γ of $\mathsf{R}[s_l]$ there exists a trace γ' of $\mathsf{R}[s_h]$ such that γ and γ' have the same observable behaviors. In other words, there exists a backward simulation from $\mathsf{R}[s_l]$ to $\mathsf{R}[s_h]$ for all non-failing terminating traces of $\mathsf{R}[s_l]$ (under the assumptions of ρ).*

(COQPROOF)

In other words, replacing a low level statement s_l, for a high level statement s_h, in a well-behaved (non-faulting) RTL program $\mathsf{R}[s_l]$, renders a program $\mathsf{R}[s_h]$ with the same observable behaviors as the original $\mathsf{R}[s_l]$ for non-failing terminating traces. Therefore, the latter can be considered a faithful implementation of the former. The notion of observable behavior in the statement above, which we inherit from the development of CompCertTSO, includes the external input/output event of the program, including system calls, failures and exit events.

We prove this theorem by induction on program executions, showing a backward trace inclusion between any $\mathcal{I}_\mathsf{L}$ trace and a trace where every subtrace corresponding to the execution of an injected statement either terminates and corresponds to a high-level statement, or is still in progress, and corresponds to a low-level statement. During

the induction step, we use the refinement property to substitute a completed subtrace of a low statement by a corresponding completed subtrace of a high statement, without affecting the ongoing traces of other threads.

We only consider non-failing traces, because failing traces represent exceptional behaviors that are not intended by the programmer. Similarly, we only consider terminated traces because we need to align low and high-level code in a one-to-one fashion. Not every code has a single point where this alignment could happen (c.f. linearizability), and hence our theorems only are concerned with states where the state can be compared between the high and low-level executing programs.

COROLLARY 5.9 (SOUNDNESS). *Let γ be a terminating, non-failing trace of the* RTL *program* $\mathsf{R}[s_l^0]\dots[s_l^n]$*, which is correctly annotated by the rely condition ρ. If for all $i \in [0,n]$, $s_l^i \preccurlyeq s_h^i$, then there exists a trace γ' of the program* $\mathsf{R}[s_h^0]\dots[s_h^n]$*, such that γ and γ' have the same observable behaviors. In other words, there exists a backward simulation from* $\mathsf{R}[s_l^0]\dots[s_l^n]$ *to* $\mathsf{R}[s_h^0]\dots[s_h^n]$ *for all terminating, non-failing traces of the former.*

(COQPROOF)

Finally, we point out that our proof rules are in no way complete. There are refinements, which while perfectly sound cannot be reached through the syntactic rules of Figure 5. As a simple example consider the rule CAS-FAIL under an environment where the location pointed by the contents of register r could never contain the value in register o. It is easy to see that in this case the *compare* would always fail, making a load of the pointer in r into the register d followed by a fence ($\mathsf{fence}; \mathsf{load}(d, r)$), equivalent to a compare-and-set ($\mathsf{cas}(d, r, o, n)$). However, we provide no way to go from the former to the latter. Moreover, in general we provide no way to go from high-level code to "equivalent" low-level versions. The purpose of our methodology is to aid the proof burden of a concurrent compiler infrastructure by making intricate pieces of code more atomic, as opposed to enabling general equivalence proofs.

5.5. Discussion

Consider the issues involved in proving a complex concurrent (racy) progarm such as a concurrent garbage collector correct. While it might be conceivable to prove the correctness of the whole collector (assuming a certain abstraction of the client code) in a single monolitic proof, we argue that by using our refinement methodology, the proof would be largely simplified by coarsening the atomicity of low-level racy pieces of code, which are only used as a service for the overall algorithm. An example of this is the bucket protocol exposed in Figures 7b and 12 to implement the write barrier, and the publication of the roots to the collector. An exemplar garbage collection invariant would require sating sophisticated properties relating the object graph, reachability of objects, their color, and the phases of the mutators, all at the same time. It is undesirable to talk about the state of the `Bucket[tid][nw]` variable when reasoning about the proofs of this kind of invariant. On the other hand, to prove the rely conditions required for our refinement of the bucket protocol, no information about the object graph, reachability of objects, or colors is necessary. Hence, our technique in a first step coarsens the atomicity of "internal" data structures, whose intermediate steps are uninteresting, and unaffected by the larger property to verify. Once this refinements have been performed, it is safe to consider these data structures as being "atomic" while performing the more challenging proof of these invariants.

We note that the conditions required to establish the soundness of refinements in the presence of visibility annotations for injected code is substantially simpler to state and prove than the invariants that capture the desired behavior of the entire runtime

system. Once a low-level statement is refined to a higher-level one, proving these more complex invariants becomes easier. Atomicity refinement is hence an important building block towards realizing tractable proofs of higher-level services and data structures.

We also note that the techniques used to prove these higher-level invariants are not constrained in any way by our refinement methodology, and do not necessarily have to be of the rely/guarantee flavor. For this reason, our formalization does not integrate a rely/guarantee logic framework to discharge the conditions. Indeed, the reader might have observed that only rely conditions are necessary to establish the soundness of our refinements; given the structure of the visibility annotations used in the refinement rules, the guarantees established by one thread invariably mirror the rely conditions of another.

6. RELATED WORK

Our strategy is reminiscent of the approach advocated in QED [Elmas et al. 2009; Elmas et al. 2010] that uses atomicity and reduction as proof tools to simplify the verification of assertions in concurrent programs. The novelty of our technique is the integration of broadly similar intuitions into a verified compiler framework, amenable to trace-based proofs. As an additional point of distinction, our techniques support the TSO memory model. Indeed, while there have been recent attempts to build verified compiler infrastructures for concurrent languages (e.g., CompCertTSO [Ševčík et al. 2011], Jinja [Lochbihler 2010], the work of Hobor *et al.* [Hobor et al. 2008]), we are unaware of other efforts focused on proof methodologies to simplify the task of verified compilation in the presence of low-level imperative and, potentially racy, concurrent code generated from high-level source and compiler-injected abstractions. Indeed, our solution does not depend on strong restrictions such as data-race freedom in either application programs or injected program fragments.

Liang *et al.* [Liang et al. 2012a] present a rely/guarantee technique to aid in the verification of a concurrent program transformer. While their solution greatly simplifies the burden of considering all possible interleavings of the concurrently executing threads, it is still very challenging to describe all the possible atomic actions that the program could perform; see [Liang et al. 2012b] for examples. Like our approach and that of [Elmas et al. 2009], their technique relies on defining preconditions and invariants, which must be discharged by means of some program logic. While it is not apparent how one might integrate their approach in a verified compiler that contain racy injected concurrent code in the absence of a refinement methodology of the kind defined here, the technique they suggest is likely to be helpful in devising rely/guarantee proofs that establish the safety of visibility annotations.

Turon and Wand [Turon and Wand 2011] present an elegant (fenced-)refinement theory, incorporating aspects of separation logic and rely-guarantee to relate ownership and atomicity based on the rely/guarantee framework of Feng [Feng 2009]. While this refinement definition is presumably sufficiently powerful for the verification of the algorithms we need to consider, the level of abstraction and the final result make it impractical to be exploited within a verifying compiler toolchain. However, their observation that "atomicity is relative to ownership" is similar to the motivation underlying our methodology.

Lochbihler [Lochbihler 2010] considers the translation phase from Java source code to Java bytecode. While a major transformation happens in this phase, none of the "injected" (as opposed to replaced one-to-one) code is concerned with fine grained atomicity issues, which is a substantial point of difference from our work.

McCreight *et al.* [McCreight et al. 2010] describe the design and implementation of a framework for certified compilation of programs in garbage-collected languages. They extend CompCert with a new intermediate language, GCminor, that provides primitives for allocation, and specifying heap roots. Their framework does not consider concurrent collection (all their collectors are "stop-and-collect"), however, and thus elides the key technical challenges addressed here.

Hobor *et al.* and Leroy *et al.* [Hobor et al. 2008; Leroy et al. 2012] define a concurrent version of CompCert's Cminor IR equipped with a concurrent separation logic. The idea is to do verified compilation for programs that have been proved correct in such a logic, which must necessarily be race-free. While likely suitable for application code, the race-free requirement is likely to be too stringent for injected code fragments like write barriers and allocators that are carefully constructed to deal with races for efficiency reasons.

Many techniques have been proposed for the verification of concurrent TSO programs: [Bouajjani et al. 2013] considers model checking, and [Alglave et al. 2013] considers verification under program rewriting. Most of these techniques are not amenable to the certification methodology we tackle in this paper.

7. CONCLUSIONS

In this paper, we present the first verifying compiler framework capable of reasoning about low-level injected, racy concurrent code cognizant of TSO relaxed memory behavior. Our primary contribution is a proof methodology that allows such code to be successively refined into higher-level atomic units whose equivalence to the original source abstraction is more easily demonstrated than a proof constructed directly on the low-level version. We have integrated our ideas within the CompCertTSO toolchain, and verified the translation of major components of a realistic concurrent GC.

Our choice of rely/guarantee-like invariants, encoded through a shallow embedding in Coq, necessitates the need for rely/guarantee program logics to discharge these assumptions. There are several existing rely/guarantee logics, for example [Dodds et al. 2009; Ridge 2010], aimed at proving these kind of programs. The latter in particular supports the TSO memory model. Fully integrating such a logic into our verified compiler toolchain is an important direction for future work necessary to enable full end-to-end verification of concurrent high-level managed languages.

Some directions for improvement in our refinement methodology include: (1) The simulation argument requires that the computations be terminating. A finer notion of simulation than the one considered in this paper could enable us to establish a finer relation between the low-level injected code, and the high-level one. In particular, this is the case for pieces of code that we know are linearizable [Herlihy and Wing 1990]. Whether by incorporating information about linearizability, or just modifying our simulation relation definition, exploring finer notions of simulation implied by our refinement is an interesting way in which we could improve our results. (2) Discharging the proof obligations related to our `@Local` annotations can be aided by an IR that makes explicit facts about memory separation and interference (perhaps through the easy expression of rely/guarantee conditions). While in this paper we do not ascribe to any particular discipline to discharge these proof obligations, designing an IR that facilitates such proofs is our most immediate research direction. (3) Targeting more relaxed memory models, such as PSO [SPARC 1994] and PowerPC [Sarkar et al. 2011] are also interesting research directions that we might consider in the future.

Other, maybe more ambitious, research directions is the verification of concurrent data structures that require low-level implementations, and are hence best implemented at the compiler level. Examples of these are monitors, and data structures that for performance reasons use architectural artifacts not present in the high-level

language (such as fences for example). Similarly, the synchronization primitives of the C11++ standard [Batty et al. 2011] could be verified under refinement, not without major modifications of our refinement rules.

References

Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering - A New Definition. In *ISCA*. 2–14.

Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *CAV*. 141–157.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL*. 55–66.

Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and Enforcing Robustness against TSO. In *ESOP*. 533–553.

Stephen D. Brookes. 1993. Full Abstraction for a Shared Variable Parallel Language. In *LICS*. 98–109.

Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. 2012. Concurrent Library Correctness on the TSO Memory Model. In *ESOP*. 87–107.

Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *ESOP*. 363–377.

Damien Doligez and Georges Gonthier. 1994. Portable, Unobtrusive Garbage Collection for Multiprocessor Systems. In *POPL*. 70–83.

Damien Doligez and Xavier Leroy. 1993. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *POPL*. 113–123.

Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Eliot E. Salant, Katherine Barabash, Itai Lahan, Yossi Levanoni, Erez Petrank, and Igor Yanover. 2000. Implementing an On-the-Fly Garbage Collector for Java. In *ISMM*. 155–166.

Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. 2010. Simplifying Linearizability Proofs with Reduction and Abstraction. In *TACAS*. 296–311.

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A calculus of atomic actions. In *POPL*. 2–15.

Xinyu Feng. 2009. Local rely-guarantee reasoning. In *POPL*. 315–327.

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *ESOP*. 353–367.

Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 596–619.

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reasoning* 43, 4 (2009), 363–446.

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Research report RR-7987. INRIA. http://hal.inria.fr/hal-00703441

Hongjin Liang, Xinyu Feng, and Ming Fu. 2012a. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*. 455–468.

Hongjin Liang, Xinyu Feng, and Ming Fu. 2012b. *A rely-guarantee-based simulation for verifying concurrent program transformations*. Technical Report. University of Science and Technology.

Andreas Lochbihler. 2010. Verifying a Compiler for Java Threads. In *ESOP*. 427–447.

Andrew McCreight, Tim Chevalier, and Andrew P. Tolmach. 2010. A certified framework for compiling and executing garbage-collected languages. In *ICFP*. 273–284.

Tom Ridge. 2010. A Rely-Guarantee Proof System for x86-TSO. In *VSTTE*. 55–70.

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *PLDI*. 175–186.

Inc. CORPORATE SPARC. 1994. *The SPARC Architecture Manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

R. Kent Treiber. 1986. *Systems Programming: Coping with Parallelism - RJ 5118*. Technical Report. IBM Almaden Research Center.

Aaron Joseph Turon and Mitchell Wand. 2011. A separation logic for refining concurrent objects. In *POPL*. 247–258.

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*. 256–271.

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-memory concurrency and verified compilation. In *POPL*. 43–54.

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22.