SANDIA REPORT
SAND2024-xxxx
Printed March 2023

Sandia National Laboratories

Correct Compilation of Concurrent C Code

John Bender

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

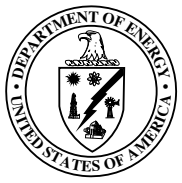
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods



ABSTRACT

The CompCert compiler [5] represents a landmark effort in program verification as both a piece of verified software and as a compiler for verified C programs. A key shortcoming of CompCert however is that it does not support multithreaded programs. Prior work to add threads to CompCert has either required major rewrites of parts of the proof [8] or only works for well synchronized programs [2]. The problem is that CompCert's backward simulation derives from a forward simulation via the determinism of the semantics of intermediate representation languages. This makes the proofs in CompCert easier but also makes them incompatible with standard models of multithreading which are non-deterministic. Here we propose an alternate formulation of CompCert's proof structure that parameterizes the existing single threaded semantics with nondeterministic behavior generated at the multithreading level. While this is an old trick where program equivalence is concerned, performing it in the context of CompCert is quite subtle. Our approach allows for expressive concurrent semantics and does not require major proof rewrites but still results in a global backward simulation for multithreaded programs.

This page intentionally left blank.

ACKNOWLEDGEMENT

Special thanks to all the project contributors Elanor Tang, Anjali Pal, Zayne Khouja, Denis Bueno and Philip Johnson-Freyd.

This page intentionally left blank.

CONTENTS

This page intentionally left blank.

LIST OF FIGURES

This page intentionally left blank.

LIST OF TABLES

This page intentionally left blank.

$$s_2 \xrightarrow{o} s_2' \wedge s_1 \sim_i s_2 \implies$$
$$\exists\, i'\ s_1',\ s_1' \sim_{i'} s_2' \wedge (s_1 \xrightarrow{o}{}^{+}\ s_1' \vee (s_1 \xrightarrow{o}{}^{*}\ s_1' \wedge i' \preceq i))$$

Figure 0-1. Backward Simulation

$$s \xrightarrow{o_1} s_1 \wedge s \xrightarrow{o_2} s_2 \implies o_1 \sim o_2 \wedge (o_1 = o_2 \rightarrow s_1 = s_2)$$

Figure 0-2. Determinacy

0.1. Introduction

CompCert derives a backward simulation from a whole-compiler forward simulation constructed from the forward simulations of each compiler pass. The backward simulation of CompCert appears in Figure 0-1. The simulation relation is denoted with $\sim$ and indexed by an i of an ordered type. Supposing the target takes a step from state s_2 there is a simulation if the resulting s_2' is related to some s_1' such that the source takes one or more steps to s_1' or it remains in the same state (i.e. $s_1 = s_1'$) and the index of the simulation relation decreases. The reason for deriving a backward simulation from a forward simulation, where the source and target steps are swapped in the definition of Figure 0-1), is that proving simulation in the forward direction is much easier. Most notably the user undertaking the proof can perform induction on the source program directly where as with a backward simulation the proof would have to work under the compiler as a function applied to the source program.

To derive the backward simulation CompCert requires two things. First it requires that the observable effects, called traces, of the source and the target are identical to preserve properties over traces of observable behavior. To see this note that in Figure 0-1, there is only one o for both the target and source steps. Second, all target semantics must be *determinate* which intuitively means that, given two steps in the target semantics for identical traces, starting in an identical state, both steps must arrive at an identical state, see Figure 0-2. On first inspection, this second property would seem to preclude the extension of CompCert's semantics with threads, which are naturally modeled with non-determinism, but we propose a semantics and proof structure that carefully balances both semantic design requirements and requires a minimum of changes to existing proofs.

0.1.1. Related Work

Prior work has attempted to address the determinacy requirement in a variety of ways. The work of [8] observed that single threaded C programs already accommodate nondeterminism from the outside world via trace inputs (e.g. a system call to rand) and moved reads and writes to memory into the observable behavior of threads. However, this conflicts directly with the trace identity

requirement and as a result backward simulation requires whole proof rewrites in some places. Separately, the work of [2] assumes that programs have been proven to be well-synchronized and thereby adopt a semantics that is equivalent to single threaded behavior but proving programs to be well-synchronized is a time consuming extra step for the user. More generally, there is extensive work on alternate forms of program equivalence that can account for nondeterminism in the context of verified compilers like the contextual equivalence proposed by [4, 9]. Contextual equivalence requires significant effort because the proof quantifies over all possible contexts, but more importantly we wish to extend CompCert and thereby avoid the onerous task of redoing all the proofs for a verified C compiler.

0.1.2. Our Work

In standard concurrent program semantics there are two sources of non-determinism. First there is the choice of threadh that will step and second there are the memory events, like reads and writes, generated by the threads. The first is derived at the level of multiple threads but the second comes from the threads themselves wherein the thread makes non-deterministic choices about things like the write a read will be paired with to provide a value. Our solution is to paramterize the single threaded semantics with any non-determinism it needs. This is an old trick! But implementing it in the context of CompCert while reusing the existing single threaded proofs is quite subtle. To leverage this approach and preserve the existing single threaded proofs for existing compiler passes we make the following contributions:

1. A prophetic multithreaded semantics that parameterizes threads with the necessary nondeterminism

2. A method for mapping between the target semantics memory and the source to guarantee a backward simulation via a presheaf construction

3. Theorems for lifting a single threaded backward simulation to a multithreaded backward simulation

4. Theorems for equivalence with a standard multithreaded semantics

5. Theorems for verification of a standard weak memory reordering pass using our method

The rest of this report proceeds as follows: in Section 2 we will give a full accounting of this approach and its subtleties; in Section 3 we give the 6 key theorems that constute the main contribution of metatheory in this workl; and in Section 3 we will cover the assumptions built into our approach about things like the operating environment.

All the mathematics in this SAND report have been formalized in a machine checkable format using the Rocq programming language and can be found here. References to the source code in what follows are relative to that link.

$$
\begin{array}{rl}
\text{Nat} & n := 0 \mid 1 \mid \ldots \\
\text{Location} & l,\, x,\, y := \ldots \\
\text{Identifier} & i,\, midx,\, tid := \ldots \\
\text{Statement} & s := \texttt{read}(l) \mid \texttt{write}(l, n) \mid \texttt{println}(l) \\
 & \quad \mid \texttt{readln}(l) \mid \texttt{fork} \\
\text{Thread} & t := \epsilon \mid s : t \\
\text{Thread State} & s := \epsilon \mid (midx, t) : ts \\
\text{Memory Event} & e := \mathsf{R}(l,\, i) \mid \mathsf{W}(l,\, n) \mid \mathsf{Print}(l,\, i) \mid \mathsf{RdLn}(l,\, n) \mid \mathsf{Noop} \\
\text{Memory} & m := \epsilon \mid (i, tid, e) : m \\
\text{Events} & h := \mathsf{print(n)} \mid \mathsf{rdln(n)}
\end{array}
$$

Figure 0-3. Exemplar Syntax

$$
\frac{t \xrightarrow{e} t' \quad \mathsf{valid}(m + e)}{(t : ts,\, m) \rightarrow (t' : ts,\, m + e)}\ \text{step-standard-thread}
$$

$$
\frac{(ts,\, m) \rightarrow (ts',\, m')}{(t : ts,\, m) \rightarrow (t : ts',\, m')}\ \text{step-standard-other}
$$

Figure 0-4. Rules For Standard Semantics

0.2. Our Approach

Here we will detail our approach. We begin by detailing the small language in which we have conducted our proofs, a traditional concurrent semantics, our concurrent semantics and use a writes-after-reads optimization as a running example. Then will describe our solution from the initial problem of preserving determinism in the single threaded semantics with the ultimate goal of developing a lifting theorem that allows us to reuse the single threaded proofs that exist in CompCert now for multithreaded programs.

0.2.1. Exemplar Semantics

We denote memory locations with l, event ids with i, and memory values with n. In Figure 0-3 we denote multithreaded-program states as a pair of a list of threads $t : ts$ and a memory state m. Threads are a list of triples of a memory index $midx$, a thread id tid, and a list of statements $s : t$, where $midx$ denotes where next memory event will come from in the thread specific set of memory for tid. Memory is a list of event ids paired with memory events: reads $\mathsf{R}(l,\, i)$, writes $\mathsf{W}(l,\, n)$, observable reads from memory $\mathsf{Print}(l,\, i)$, observable reads into memory from the environment $\mathsf{RdLn}(l,\, n)$ and noops, Noop. In the case of reads and prints the i denotes the identifier that the

$$\frac{m \preceq m'}{(ts, m) \longrightarrow (ts, m')}\ \text{step-prophecy-prophesy}$$

$$\frac{t \rightarrow_m t'}{(t : ts, m) \longrightarrow (t' : ts, m)}\ \text{step-prophecy-thread}$$

$$\frac{(ts,\ m) \longrightarrow (ts',\ m')}{(t : ts,\ m) \longrightarrow (t : ts',\ m')}\ \text{step-prophecy-other}$$

Figure 0-5. Prophetic Semantics

```
Fixpoint reorder_writes_reads_pairs (m : list (mem_rec * mem_rec)) : list (mem_rec * mem_rec):=
  match m with
  | [] => []
  | (ev1, ev2) :: m' =>
      (match ev1, ev2 with
      | (i1, tid1, mem_ev_read l1 iw n1), (i2, tid2, mem_ev_write l2 n2) =>
          if negb (ident_eq l1 l2) && (Nat.eq_dec tid1 tid2) (* different locations, same thread *)
          then (ev2, ev1)
          else (ev1, ev2)
      | _, _ => (ev1, ev2)
      end) :: reorder_writes_reads_pairs m'
  end.
```

Figure 0-6. Read-Write Reordering

read takes its value from. We sometimes omit the related write for simplicity. Observable traces are denoted with observable events, $\mathsf{print}(n)$ and $\mathsf{rdln}(n)$, in lists ev. Thread state is a pair of the memory index and list of statements.

A traditional multithreaded semantics has two rules as in Figure 0-4 step-standard-thread and step-standard-other, allowing the first thread to step and generate a memory even to be validated by the memory model and allowing some other thread to step, respectively.

Our prophetic multithreading semantics has three rules in Figure 0-5: extend the memory with non-deterministic choices (step-prophecy-prophesy) according to the preorder which we will detail in Section 0.2.4.1, allow the first thread to do some work with the current memory state (step-prophecy-thread), and choosing another thread to do some work (step-prophecy-other).

The single threaded semantics not depicted resolves the event in the memory m at $midx$ as matching the shape of the statement at the head of the thread.

The mechanization of these semantics can be found in the Coq development file `concurrency/Semantics.v`.

0.2.2. Example Optimization

In Figure 0-6 we provide an example mapping from an optimized program where the writes in a thread are reordered after reads in the same thread when their locations are different to the original

ordered program. That is, the mapping swaps the reads back before the write to match the source program. We also define a predicate that will inform the preorder discussed later in Section 0.2.4.1 where we will see why reordering is a particularly important optimization where the preorder is concerened.

0.2.3. Non-determinism

Traditionally thread steps in concurrent semantics take the form of the step-standard-thread rule in Figure 0-4. The step-standard-thread rule chooses the thread t at the head of the list to execute. For a thread step, the thread semantics generates an event ev for an expression that operates on memory. For example, a read of the variable x with a value 1. Then the valid predicate ensures the event is possible, in this case that there is some visible write to x of the value 1. What's important is that the thread is making a non-deterministic guess about the possible values for the variable x. In the context of CompCert this makes a traditional single threaded semantics unworkable.

In contrast to prior work, our solution is to invert this relationship and parameterize the single threaded semantics with a memory state that contains the necessary non-deterministic choices for the thread to operate deterministically. We denote this with $\rightarrow_m$ in step-prophecy-thread in Figure 0-5. The idea is that the memory state should include enough non-deterministic choices for upcoming memory operations so that we can recover a deterministic thread semantics.

Of course this is a small conceptual change and an old trick in program equivalence, but it has important implications for simulation proofs in CompCert. To start, it introduces an existential quantifier into the backward simulation. In Figure 0-7 we have now introduced the memory state m_2 for the target and m_1 for the source. Intuitively, for any memory state that works for a target step we need to find at least one memory state for a step in the source.

0.2.4. Memory Mapping

Our solution to the problem of determinacy raises yet another issue as this new existential would constitute a major update to every proof in CompCert because we would need to make a choice for the newly quantified source memory state. To solve this we utilize a skolem function to make the existential choice by defining a mapping from the target memory to the right source memory. We use this mapping from target memories to construct an *indexed backward simulation* which is indexed over a preordered type for memory states.

The idea is that for a compiler pass that manipulates memory, this mapping will transform the memory of the target program so that it's possible to find a step for the source program that matches the target. For example, if a compiler pass were to reorder a write to the variable x and a read to the variable y the attendant mapping would switch those operations back in the memory state for the source step as in Figure 0-6.

However, since we will define such a mapping for every compiler pass that manipulates memory operations within the program, and each of those will be composed into a global mapping to match the final composition of every compiler pass, we won't be able to inspect the contents of the final

$$s_2 \xrightarrow{o}_{m_2} s_2' \wedge s_1 \sim_i s_2 \implies$$
$$\exists\, m_1\, i'\, s_1',\ s_1' \sim_{i'} s_2' \wedge (s_1 \xrightarrow{o}_{m_1}{}^{+} s_1' \vee (s_1 \xrightarrow{o}_{m_1}{}^{*} s_1' \wedge i' \leq i))$$

Figure 0-7. Backward Simulation with Memory State

$$s_2 \xrightarrow{o}_{m_2} s_2' \wedge s_1 \sim_i s_2 \implies$$
$$\exists\, i'\, s_1',\ s_1' \sim_{i'} s_2' \wedge (s_1 \xrightarrow{o}_{f\, m_2}{}^{+} s_1' \vee (s_1 \xrightarrow{o}_{f\, m_2}{}^{*} s_1' \wedge i' \leq i))$$

Figure 0-8. Indexed Backward Simulation

$$\mathsf{monotonic}_{p_1,p_2}\ f \triangleq \forall m_1\, m_2,\ m_1 \leq_{p_1} m_2 \implies f\ m_1 \leq_{p_2} f\ m_2$$
$$\mathsf{scottcont}_{p_2}\ f \triangleq \forall m_1\, m_2, p_2\, m_1 \implies f\ (m_1\ +\!+\ m_2) = f\ m_1\ +\!+\ f\ m_2$$

Figure 0-9. Monotonicity & Scott Continuity

mapping. Thus, we must provide some general constraints for each such mapping to satisfy such that we can construct a global multithreaded backward simulation using the composed mapping. Similarly, those constraints must also compose so that global mapping will carry them forward.

The first such condition is that the mapping should be monotonic with respect to a preorder over memories. The preorder will capture the idea that memories can be ordered by how many "behaviors" as steps they permit and thus, if the mapping is monotonic it should also always result in more behaviors which is exactly what we want when proving a backward simulation. Roughly, the source memory we are mapping too should permit more steps in the source program.

The second is that the mapping should distribute over subsets ordered according to the definition of the preorder making it Scott Continuous [7]. As the preorder defines the idea of extending a memory with more behaviors the mapping should be able to operate on those extensions without concern for the smaller memory contents.

0.2.4.1. Predicated Preorder

There are three critical parts of our metatheory that depend on the definition of the preorder:

First, it must be strong enough support the monotonicity of steps in the single threaded semantics. This is what it means for a "larger" memory to support more behaviors.

Second, it must be weak enough that it admits a backward simulation from a standard concurrency semantics to our prophetic semantics. That is, when a thread in a standard semantics generates

a memory event the preorder has to be weak enough to allow any such event to be added to the prophetic memory in (step-prophecy) in Figure 0-5.

Third, it must be calibrated carefully to allow compiler pass verifiers to verify many optimizations under the montonicity constraint. In this case it must both constrain the memories being ordered in the antecedent and also allow the memory mapping freedom to change memory in the consequent.

A naive approach is to ordered memories is through direct extension:

$$m_1 \leq m_2 := \exists\, ext, m_2 = m_1 {+}{+}\, ext$$

This definition satisfies the first and second properties directly. In the first case, any step with m_1 will also be possible with m_2 because the same event that enabled the step with m_1 is also present in m_2. In the second case, the prophetic semantics can always guess the same event that the thread generated in the standard semantics by extending to including it and matching memories in the program state.

But consider the reordering optimization in Figure 0-6 and the following candidate memories that represent a counter example.

$$\begin{aligned}[\mathsf{R}(x)] \leq [\mathsf{R}(x), \mathsf{W}(y,\, n)] \implies & f\ [\mathsf{R}(x)] \leq f\ [\mathsf{R}(x), \mathsf{W}(y,\, n)] \\ \implies & [\mathsf{R}(x)] \leq [\mathsf{W}(y,\, n), \mathsf{R}(x)]\end{aligned}$$

Here we write f for `reorder_writes_reads_pairs`. While it's possible to find an extension to order the unmodified memory state, the mapping will fail to preserve the ordering. The key observation we can make here is that the memory mapping will produce different values based on the amount of information in the target memory state. Thus we parameterize the preorder with a predicate intended to ensure that it's only possible to order memory states with enough information for the mapping to operate.

$$m_1 \leq_p m_2 \triangleq \begin{cases} p\ m_2 \wedge \exists ext,\, m_2 = m_1\ {+}{+}\, ext & \text{if } p\ m_1 \\ m_1 = m_2 & \text{otherwise} \end{cases}$$

Intuitively, we expect an extension when both of the memories match the predicate and otherwise we are "stuck" with the existing memory.

For the counterexample above we define $p\ m \triangleq \mathsf{even}(\mathsf{length}\ m)$ and this rules out any memories that lack right shape to be considered in the preorder, in particular odd events sequences like $[\mathsf{W}(x,\, v)]$. That is, when ordered guesses are made in the prophetic semantics they must contain enough information for the mapping to operate properly. Concretely, when encountering a read the next event must be available to see if it's a write that should be reordered.

Note, not depicted here we additional require that if m_1 is valid according to the memory model of the semantics the m_2 must also be valid. This ensures that the memory mapping always preserves validity for any extension it makes and will be relevant in the proof of lifting for single threaded to multithreaded backward simulation.

0.2.4.2. Montonicity and Predicated Preorder

Importantly though this new defintion breaks the natural form of composition for monotonicity that comes with a uniform preorder. To address this we provide a new theorem of composition for the monotonicity of mappings with the notable caveat that the second predicate must be consistent among them. In practice the second predicate will simply be 'True' as we expect at the top level the ordering will require no constraints (since there is no further mapping to be done).

$$\mathsf{monotonic}_{p_1,\,p_3}\ f \implies \mathsf{monotonic}_{p_2,\,p_3}\ g \implies \mathsf{monotonic}_{(\lambda m,\, p_1\, m\ \wedge\ p_2\,(f\, m)),\,p_3}\ (g \circ f)$$

We give a proof in Section 0.3.4.

Further we must restrict the predicates such that they cannot rule out certain memory events and thereby break the second critical constraint on the preorder by preventing the prophetic semantics from guessing an event generated by the axiomatic semantics.

$$\mathsf{strict}_p \triangleq p\ \varnothing = \top$$
$$\mathsf{safe}_p \triangleq \forall m\, e_1\, e_2,\ e_1 \in m \implies p\ m = p\ ((m/\{e_1\}) \cup \{e_2\})$$

Proposition 0.2.1.

$$\forall f, \mathsf{safe}\ p_1 \implies \mathsf{safe}\ p_2 \implies \mathsf{safe}\ (\lambda m, p_1\ m \wedge p_2\ (f\ m))$$

Here we require that it be possible to swap out any event in a given memory without changing the outcome the predicate's result. More generally we should show that such a safety property composes along with monotonicty so that we will have a safe composed predicate when proving the backward simulation from standard to prophetic semantics.

0.2.4.3. Scott Continuity

We additionally require that mappings be Scott continuous as in Figure 0-9 for the directed set of memory extensions defined by the attendant predicated preorder. This is necessary to for proving monotonicity because there we assume an ordered memory which means an extension and then must show that the mapping will operate properly on both the original memory and its extension.

We give a proof of this property for the example reordering mapping in Section 0.3.5.1.

0.3. Metathory

Here we detail how our approach supports the key metatheorems outlined in the introduction. We give each theorem and sketch of the proof along with a reference to their mechanized proofs.

0.3.1. Indexed Forward to Backward Simulation

Lemma 0.3.1. *Given forward simulation indexed by a preorder type adn mapping we can derive a backward simulation for the same type.*

Proof. See the Coq development file `common/Presheaf.v` □

0.3.2. Lifted Backward Simulation

Theorem 0.3.2. *If for any state and step from that state in the single threaded target semantics we have an indexed backward simulation to a single threaded step in the source semantics then we can demonstrate a multithreaded backward simulation from the target to the source.*

Proof Sketch. By assumption we have that there is a step in the multithreaded target program from some memory state m_2 (we elide discussion of non-memory state here for brevity) and we must show that we can construct a step in the multithreaded source program from a state related by a simulation relation.

We can construct the simulation relation using the single threaded simulation relation for each thread and by assigning the memory state of the source to be $f\ m_2$ for the memory mapping in the indexed backward simulation.

We proceed by induction on the step of the multithreaded target semantics. In the case that a thread is stepping we must show that the thread can step from a state in the simulation relation and that the state it arrives at is also in the relation. First, we note that the target will have stepped to some new memory state m_2' and that $m_2 \preceq m_2'$ according to step-prophecy-prophesy in Figure 0-5. Second, we must have that $f\ m_2'$ and m_2' will be related memory states according to our simulation relation. Then we use the indexed backward simulation to derive a single threaded step in the source semantics from the memory state $f\ m_2'$. We must also show that $f\ m_2'$ is valid according the given memory model which follows from the definition of the preorder and that fact that the memory mapping must preserve the validity of its input. □

In the case where the semantics chooses a thread not at the head of the list the inductive hypothesis applies and we are done.

0.3.3. Equivalence with Standard Semanatics

0.3.3.1. Axiomatic semantics simulated by Prophetic Semantics

Theorem 0.3.3. *For every step in the standard Axiomatic Semantics there is a matching step in the Prophetic Semantics with matching states and observations.*

Proof Sketch. First, note that the program is identical in both semantics. Second, note that we define a simulation relation that requires all memory extensions in the prophetic memory sate comport with statements in the program and that the prophetic memory state is an extension of the axiomatic memory state.

We proceed by induction on the Axiomatic step.

In the case where a thread steps and generates an event, the Prophetic Semantics either has enough events from previous guesses which must match the program according to the simulation relation which means it can already take a thread step or it first guesses a memory extension that comports with the predicate p for the attendent preorder. Since p is safe, the next event can match the next statement in the thread and the step procees.

The case of an different thread stepping follows from the inductive hypothesis. □

The mechanization can be found in the Coq development file `concurrency/Simulation.v`.

0.3.3.2. Prophetic Semantics simulated by Axiomatic Semantics

Theorem 0.3.4. *If we assume that a series of guesses is bounded by a decreasing order, one guess per thread without events to execute, then for every step in the Prophetic Semantics there is a matching step in the Axiomatic Semantics with matching states and observations.*

Proof Sketch. We proceed by induction on the Prophetic step.

In the case guessing a new memory state, we allow the Axiomatic Semantics to sit still with the assumed decreasing order or the decreasing order is already at the least element and the guess step is not allowed producing a contradiction.

In the case where a thread steps the Axiomatic Semantics can match according to the event that was used to step the thread in the Prophetic Sematics and for which we know the memory will be valid because the Prophetic Semantics requires all guesses to be valid.

The case of a different thread stepping follows from the inductive hypothesis. □

The mechanization can be found in the Coq development file `concurrency/Simulation.v`.

0.3.4. Composition of Predicated Preorder

Theorem 0.3.5. *Given two functions f and g which are monotone over the preorders parameterized by the predicates p_1 and p_3 for f and p_2 and p_3 for g then $g.f$ is monotone with respect to the predicates $p_1 m \&\& p_2(fm)$ for any memory m.*

Proof Sketch. We proceed by cases on each instance of monotonicity.

For any m_1 and m_2, $m_1 \leq_{p_1} m_2$ implies $f\ m_1 \leq_{p_3} f\ m_2$. For any m_1 and m_2, $m_1 \leq_{p_2} m_2$ implies $g\ m_1 \leq_{p_3} g\ m_2$. Assume $m_1 \leq_{\forall m, p_1 \&\& p_2(fm)} m_2$ Show $g\ (f\ m_1) \leq_{p_3} g\ (f\ m_2)$

Note that each predicate is a boolean valued function and thefefore the outcome is decidable.

We proceed by cases on $p_1\ m_1 \&\& p_2\ (f\ m_1)$. If it's not true $m_1 = m_2$ so whether $p_3\ m_1$ is true or not the ordering follows trivially.

If it is true then we have that $p_1\ m_1$ and $p_2\ (f\ m_1)$, also $p_1\ m_2$ and $p_2\ (f\ m_2)$ and that m_2 is an extension of m_1.

From that we can derive that $f\ m_1 \leq_{p_3} f\ m_2$ from the first assumption instantiated with m_1 and m_2. We can also derive that $g\ (f\ m_1) \leq_{p_3} g\ (f\ m_2)$ from the second assumption instantiated with $f\ m_1$ and $f\ m_2$ as required. □

0.3.5. Reordering Reads and Writes

0.3.5.1. scott continuous

Theorem 0.3.6. *If a memory m_1 satisfies the ordering predicate* even(length m_1) *then $f\ (m_1 + +m_2) = f\ m_1 + +f\ m_2$.*

Proof Sketch. Follows from the fact that f will only fail to deal with a tail element when the length of m_1 is odd.

□

0.3.5.2. Monotonicity

Theorem 0.3.7. *The reordering optimization in Figure 0-6 is monotonic with respect to the order predicated by* even(length$m$) *and $\top$ and the validity predicate of the X86 memory model. That is, for any memories m_1 an m_2 we have that if $m_1 \leq_{even} m_2$ then $f\ m_1 \leq_\top f\ m_2$.*

Proof Sketch. The validity requirement follows from the fact that X86 allows writes to be reordered with reads to different locations in the same thread. Then if the even length predicate does not hold for the assume ordering of memories the memories are equal and montonicity follows directly.

We must show that if the ordered memories are of even length then fm_2 is an extension of fm_1. Since we know that m_2 is an extension of m_1 according to the antecedent its enough to show that $f(m_1{+}{+}ext)$ is an extension of fm_1, by scott continuity we know that $f\ (m_1{+}{+}ext) = f\ m_1{+}{+}f\ ext$ and thus $f\ ext$ is the extension of $f\ m_1$ as required. □

The mechanization can be found in the Coq development file `concurrency/Optimizations.v`.

```
Fixpoint reorder_writes_reads_opt (ev_op : option mem_rec) (m : mem) : mem :=
  match ev_op, m with
  | None, ev1 :: m' => reorder_writes_reads_opt (Some ev1) m'
  | Some ev1, [] => [ev1]
  | Some ev1, ev2 :: m' =>
      match ev1, ev2 with
      | (i1, tid1, mem_ev_read l1 iw n1), (i2, tid2, mem_ev_write l2 n2) =>
          if negb (ident_eq l1 l2) && (Nat.eq_dec tid1 tid2) (* different locations, same thread *)
          then ev2 :: reorder_writes_reads_opt (Some ev1) m'
          else ev1 :: reorder_writes_reads_opt (Some ev2) m'
      | _, _ => ev1 :: reorder_writes_reads_opt (Some ev2) m'
      end
  | _, _ => m
  end.
```

Figure 0-10. Alternate Optimization

0.4. Assumptions and Implications

0.4.1. Operating Environment

All concurrent semantics embed a few key assumptions about their operating environment. First, that shared memory is only ever manipulated by threads within the same program. Second, that forking a new thread acts as a hard syncrhonization between the forked thread and memory operations from before the fork. Third, we also assume that there is a clear distinction between thread local memory and shared memory, the former of which should not appear in the memory representation to make compiler pass verification easier.

The first two assumptions are embedded in our semantics. In the first case because only the program can manipulate the memory state. In the second case because a fork generates a whole new, thread specific memory.

We have not yet expressed nor explored the need for local memory manipulations where our model of concurrency is concerned. Memory operations on non-shared memory should not appear in the memory representation and thus the attendant memory mapping should be the identity function. However this requires some guarantee about the fact that memory, known to be local like the stack frame for a function, will not be manipulated by other threads.

0.4.2. Predicated preorder

An important caveat for the predicated preorder in our approach is that each mapping, and thus each optimization, must work within a finite window of memory events. To see why consider the following modification to the the reordering mapping in Figure 0-10.

Consider the second, more natural recursive implementation, operating one event at a time. As before we can't find an extension to fix the reordering issue without a predicate but here it's also hard to say what the predicate should be to ensure the mapping has enough information to proceed because it may reorder a read past every write in the entire memory without a boundary. For a

$$\text{safe } s \triangleq \forall s', s \xrightarrow{\varnothing} s' \implies \text{final } s' \vee (\exists\, t\ s', \xrightarrow{t} s'')$$

Figure 0-11. Alternate Optimization

program where with nothing but writes after a read this means we could never guess enough events to make the mapping work properly and thus monotonicity would not be preserved.

$$\begin{aligned}[\mathsf{R}(x), \mathsf{W}(y,\, n)] \leq [\mathsf{R}(x), \mathsf{W}(y,\, n), \mathsf{W}(z,\, n)] &\implies f[\mathsf{R}(x), \mathsf{W}(y,\, n)] \leq [\mathsf{R}(x), \mathsf{W}(y,\, n), \mathsf{W}(z,\, n)] \\ &\implies [\mathsf{W}(y,\, n), \mathsf{R}(x)] \leq [\mathsf{W}(y,\, n), \mathsf{W}(z,\, n), \mathsf{R}(x)]\end{aligned}$$

Again, the result is not ordered because there's no viable extension. It seems clear that some finite amount of consideration is required such that we can define a predicate that will allow finite guesses and still preserve monotonicity.

0.4.3. Safety and Fairness

CompCert defines safe states for a semantics as in Figure 0-11. The key idea is that from any $safe$ state, if we can take zero or more steps, the program is in a final state or another step can be taken. This idea of safety is based on the important work of [10] where bad program states are represented by a "stuck" state, ie where no step rules apply. In CompCert undefind behavior of C programs is captured as a "stuck" states. Thus, if a program can always take a step from a safe state and assuming determinism, it can never perform undefined behavior.

In the context of a multithreaded program the notion of "stuck" becomes much more subtle. For example, it could be that a particular thread is stuck but there are other threads that can execute indefinitely so the multi-threaded program is safe in the sense of not being stuck.

Most importanty, the backward simulation that CompCert proves for single threaded programs which we use to prove the multithreaded backward simulation in Section 0.3.2 requires that we demonstrate the state we've instantiated the single threaded backward simulation with is in fact safe. Our approach is to derive this fact from the assumption of a multithreaded safe state. Intuitively, a safe state in the multithreaded context should imply that the derived state for each thread is safe.

Initially, this requires at least some notion of fairness because, as described the existing definition of safety in CompCert does not guarantee that any given thread can step even when they are enabled. However there are additional issues which require addressing.

Most obviously, we give the semantics a finite memory to operate, without a new extension of the memory with memory events for the thread from the concurrent semantics the thread may run out of memory events to execute. Taken with the observation that the predicate which defines final states does not have access to the memory state since it's part of the semantics if the thread runs out of memory events it might not be in a final state and no able to step.

0.4.4. Proof burden for pass verifiers

It's important to note that this approach does obligate the compiler pass author to provide a mapping that can determine source memory states from target memory states. We believe that this extra side-obligation works out rather naturally within existing proofs and requires few modifications even when a pass does modify memory.

Further we have not yet been able to test this approach with real compiler passes so direct experimental evidence of how much modification of existing proofs is required is still unknown. We expect that this should minimal, confined largely to the extension of intermediate semantics to incorporate a memory parameter. Otherwise the mapping proofs should have no impact on the compiler pass beyond where that mapping pertains to their correctness with respect to a given memory model (e.g. if the optimization is not allowed).

REFERENCES

[1] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014.

[2] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W Appel. Verified compilation for shared-memory c. In *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 23*, pages 107–127. Springer, 2014.

[3] Soham Chakraborty and Viktor Vafeiadis. Grounding thin-air reads with event structures. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.

[4] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. *ACM SIGPLAN Notices*, 53(4):646–661, 2018.

[5] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[6] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: X86-tso. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] Scott Continuity. Scott continuity, 2024. [Online; accessed 1-October-2024].

[8] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *Journal of the ACM (JACM)*, 60(3):1–50, 2013.

[9] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. Conditional contextual refinement. *Proceedings of the ACM on Programming Languages*, 7(POPL):1121–1151, 2023.

[10] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Technical Library	1911	sanddocs@sandia.gov

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop
1	L. Martin, LDRD Office	1910	0359

This page intentionally left blank.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.