Correct Compilation of Concurrent C Code

John Bender
jmbende@sandia.gov
Sandia National Laboratories
Los Angeles, USA

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 [7] 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.

CCS Concepts: • **Theory of computation → Program semantics**; • **Software and its engineering → Compilers**.

Keywords: Concurrency, Memory Models, Compilers, Verification

ACM Reference Format:
John Bender. 2024. Correct Compilation of Concurrent C Code. In *Proceedings of the Workshop Dedicated to Jens Palsberg on the Occasion of His 60th Birthday (JENSFEST '24), October 22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3694848.3694856

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 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 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.

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

Figure 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 2. Determinacy

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 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 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.

JENSFEST '24, October 22, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1257-9/24/10
https://doi.org/10.1145/3694848.3694856

2 Related Work

Prior work has attempted to address the determinacy requirement in a variety of ways. The work of [7] 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, 8]. 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.

3 Our Work

Traditionally thread steps in concurrent semantics take the form of the step rule in Figure 3. Program states are list of threads $t : ts$ and a memory state m. The step rule chooses the thread t at the head of the list to execute. Usually there is another rule (not-depicted) that chooses from ts. For a thread step, the thread semantics generates an event for an expression that operates on memory. For example, a read of the variable x with a value 1. Then the validate 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.

Our solution is to 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 $\rightarrow_{(m+ev)}$ in step-prophecy in Figure 3. 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 but it has important implications for simulation proofs in CompCert. To start, it introduces an existential quantifier into the backward simulation. In Figure 4 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.

Here though we have yet another issue, this 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 we call a memory mapping to construct an *indexed backward simulation* which is indexed over a preordered type for memory states which we discuss further in Section 4.

$$\frac{t \xrightarrow{\text{ev}} t' \quad \text{valid}(m + \text{ev})}{(t : ts, m) \rightarrow (t' : ts, m + \text{ev})}\ \text{step}$$

$$\frac{t \rightarrow_{(m+\text{ev})} t' \quad \text{valid}(m + \text{ev})}{(t : ts, m) \rightarrow (t' : ts, m + \text{ev})}\ \text{step-prophecy}$$

$$\frac{t \rightarrow_{m'} t' \quad \text{valid}(m') \quad m \leq m'}{(t : ts, m) \rightarrow (t' : ts, m')}\ \text{step-ordered-prophecy}$$

Figure 3. Different Thread Step Semantics

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

Figure 4. 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 5. Indexed Backward Simulation

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.

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

4 Lifting to Multithread Backward Simulation

At this point we have a way to construct single threaded backward simulations for CompCert using a memory mapping but the ultimate goal is constructing a multithreaded backward simulation. Notably, we have not placed any constraints on the mapping that generates our source memory

state. When we leverage the single threaded backward simulation we won't know anything about the mapping and even if we did it would be a composite from every compiler pass. Thus we must constrain the behavior of the mapping more generally while still allowing enough flexibility to accommodate many optimizations.

Intuitively, a memory state generated by the mapping should only ever allow more behaviors of the source program in keeping with the backward simulation. Thus we require that the memory mapping be monotonic in a preorder over memory state. We leave the preorder and the precise formulation of the memory state abstract for now. Similarly, for a single threaded source step to accommodate such a memory mapping we require that the single threaded semantics be monotonic in the same preorder. Intuitively, threads should not refuse to take a step when given nondeterministic choices from a more permissive memory state.

Finally, we adopt a modified rule for thread steps, step-ordered-prophecy in Figure 3. The idea is that the semantics can always make new non-deterministic guesses as long as they permit more behaviors[1]. This modification has a particularly practical bent as well that will be clear in the proof sketch for our main proposition.

Theorem 4.1. *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 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-ordered-prophecy in Figure 3. Second, we must have that $f\, m_2'$ and m_2' will be related memory states according to our simulation relation.

Then, since we are stepping from $f\, m_2$ to $f\, m_2'$ in the source we must show that $f\, m_2 \preceq f\, m_2'$ but that follows from the monotonicity of f and the fact that $m_2 \preceq m_2'$. 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 show that the same single threaded step is possible with $f\, m_2'$ but then this follows from monotonicity of the single threaded semantics.

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

[1]Note that the preorder has to be constructed carefully not to allow memory events already "used" by threads to be replaced by a new guess because that would allow non-standard behaviors. For example if we suddenly changed the value at an existing visible write.

$$m_1, m_2 : \text{list ev}$$
$$m_1 \preceq m_2 \triangleq \text{length } m_1 \leq \text{length } m_2$$

Figure 6. An Example Order

$$f\, m \triangleq \begin{cases} \text{ev} : f\, m' & \text{if } m = \text{ev} : \text{F} : m' \\ \text{ev} : f\, m' & \text{if } m = \text{ev} : m' \\ nil & \text{if } m = nil \end{cases}$$

Figure 7. Example Mapping

$$[\text{R}(y), \text{R}(x)] \preceq [\text{R}(y), \text{F}] \implies f\,[\text{R}(y), \text{R}(x)] \preceq f\,[\text{R}(y), \text{F}]$$
$$\not\Longrightarrow [\text{R}(y), \text{R}(x)] \preceq [\text{R}(y)]$$

Figure 8. Montonicity Example

5 Future Work

Previously we left the two important definitions abstract, the preorder and the memory state representation. These definitions work together: for the preorder we must pick a notion of ordering that does not prevent us from correctly ordering the memory mapping for desired compiler passes and the memory representation dictates what data we can use to order memory states. For example, a naive approach is to directly adopt the graph structures and validity predicates from the axiomatic approaches in the memory models literature [1, 6] and order memory states by the length of a list that stands in helpfully for a set, see Figure 6.

This turns out to work quite well for many optimizations including any optimization that removed memory operations (and thus would require a mapping to add them back in) like redundant read elimination. Clearly though passes that add shared memory events would require a different order. For example a compiler pass that inserts fences [7] like f in Figure 7 would not produce a monotonic memory mapping as demonstrated by the counterexample at the bottom of Figure 8.

However this counterexample is instructive! The mapping is removing a memory event (the fence) but it is increasing the possible behaviors of the source program by making the memory representation more permissive. Thus our current

task is finding a memory representation and preorder that permits many optimizations, fits with our intuition about how memory representations should be ordered by the behavior of programs, and is simple enough work with for compiler pass authors. We are currently exploring extensions of the standard memory model graphs and also event structures like those of [3].

Finally, we wish to extend the validation of our work through more compiler optimizations built for our toy example languages and then also for CompCert proper.

References

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

[2] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W Appel. 2014. 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*. Springer, 107–127.

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

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

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

[6] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: X86-TSO. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics* (Munich, Germany) *(TPHOLs '09)*. Springer-Verlag, Berlin, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

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

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