Verifying Optimizations for Concurrent Programs*

William Mansky and Elsa L. Gunter

Department of Computer Science, University of Illinois at Urbana-Champaign, Thomas M. Siebel Center, 201 N. Goodwin, Urbana, IL 61801-2302, USA
`{mansky1,egunter}@illinois.edu`

Abstract

While program correctness for compiled languages depends fundamentally on compiler correctness, compiler optimizations are not usually formally verified due to the effort involved, particularly in the presence of concurrency. In this paper, we present a framework for stating and reasoning about compiler optimizations and transformations on programs in the presence of relaxed memory models. The core of the framework is the PTRANS specification language, in which program transformations are expressed as rewrites on control flow graphs with temporal logic side conditions. We demonstrate our technique by verifying the correctness of a redundant store elimination optimization in a simple LLVM-like intermediate language, relying on a theorem that allows us to lift single-thread simulation relations to simulations on multithreaded programs.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases optimizing compilers, interactive theorem proving, program transformations, temporal logic, relaxed memory models

Digital Object Identifier 10.4230/OASIcs.WPTE.2014.15

1 Introduction

Program verification relies fundamentally on compiler correctness. Static analyses for safety or correctness in compiled languages depend implicitly on the fidelity of the compiler to some abstract semantics for the language, but real-world compilers rarely reflect these theoretical semantics [17]. The optimization phase of compilation is particularly error-prone: optimizations are often stated as complex algorithms on program code, with only informal justifications of correctness based on an intuitive understanding of program semantics. Formal methods researchers have devoted considerable effort to verifying these optimizations, either on a program-by-program basis (the translation validation approach [13]), or by general proof of correctness for all possible inputs (the approach taken, for instance, in the CompCert verified C compiler [7]). The problem is only aggravated in the presence of concurrency. Insufficiently analyzed optimizations may result in unreliable execution of concurrent code; compiler writers may even end up having to limit the scope and complexity of the optimizations they develop, in the absence of a method to demonstrate the safety of their optimizations.

In this paper, we present a new methodology for stating and verifying the correctness of compiler optimizations and transformations in the presence of concurrency, centered around a domain-specific language for specifying optimizations as transformations on program graphs

* This material is based upon work supported in part by NSF Grant CCF 13-18191. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'14).
Editors: Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba; pp. 15–26
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

with temporal logic side conditions. This language, PTRANS, has been formalized in the Isabelle proof assistant [12], so that optimizations expressed in PTRANS can be proved correct with the assistance of state-of-the-art theorem-proving tools, as well as an executable semantics allowing specifications to serve as optimization prototypes. As a proof of concept, we use PTRANS to express and verify an optimization under several different concurrent memory models. Ultimately, we hope that the approach outlined in this paper will assist both formal verifiers and compiler writers in creating complex, concurrency-safe optimizations.

2 The PTRANS Specification Language

2.1 PTRANS: Adapting TRANS to Parallel Programs

The basic approach of the PTRANS specification language is that set out by Kalvala et al. in TRANS [4]: optimizations are specified as rewrites on program code in the form of control flow graphs (CFGs), with side conditions given in temporal logic. The syntax of PTRANS is given by the following grammar:

$$\begin{array}{rcl} A & ::= & \mathsf{add\_edge}(n, m, \ell) \mid \mathsf{remove\_edge}(n, m, \ell) \mid \mathsf{split\_edge}(n, m, \ell, p) \\ & \mid & \mathsf{replace}\ n\ \mathsf{with}\ p_1, ..., p_m \\ \varphi & ::= & \mathsf{true} \mid p \mid \varphi \wedge \varphi \mid \neg\varphi \mid A\ \varphi\ \mathcal{U}\ \varphi \mid E\ \varphi\ \mathcal{U}\ \varphi \mid A\ \varphi\ \mathcal{B}\ \varphi \mid E\ \varphi\ \mathcal{B}\ \varphi \mid \exists x.\ \varphi \\ T & ::= & A_1, ..., A_m\ \text{if}\ \varphi \mid \text{MATCH}\ \varphi\ \text{IN}\ T \mid T\ \text{THEN}\ T \mid T \,\Box\, T \mid \text{APPLY\_ALL}\ T \end{array}$$

The atomic actions A include add_edge and remove_edge, which add and remove (ℓ-labeled) edges between the specified nodes; split_edge, which splits an edge between two nodes, inserting a new node between them; and replace, which replaces the instruction at a given node with a sequence of instructions, adding new nodes to contain the instructions if necessary. Kalvala et al. have shown that a wide range of common program transformations can be expressed using these basic rewrites. The arguments to the atomic actions represent nodes and instructions in the program graph, but may contain *metavariables* that are instantiated to program objects when the rewrites are applied.

At the top level, a transformation T is built out of conditional rewrites combined with *strategies*. The term $A_1, ..., A_m$ if φ is the basic pairing of one or more rewrites with a first-order CTL side condition, which may include the forward until-operator $\mathcal{U}$, its backward counterpart $\mathcal{B}$, and quantifiers over the metavariables appearing in its atomic predicates p. The expression MATCH φ IN T provides an additional side condition for a set of transformations, and also allows metavariables to be bound across multiple rewrites. The THEN and □ operators provide sequencing and (nondeterministic) choice respectively, and APPLY_ALL T recursively applies T wherever possible until it is no longer applicable to the graph under consideration.

2.2 Concurrent Control Flow Graphs

The TRANS-style approach depends fundamentally on a notion of control flow graph (CFG). The atomic rewrites are rewrites on CFGs, and the CTL side conditions are evaluated on paths through CFGs. Thus, we require a concurrent analogue to the CFG in order to extend the approach to the concurrent setting. The particular model used here, adapted from the work of Krinke [5], is the threaded control flow graph (tCFG). In our framework, a tCFG is simply a collection of non-intersecting CFGs, one for each thread in a program. Formally:

▶ **Definition 1.** A *CFG* is a labeled directed graph $(N, E, \mathit{Start}, \mathit{Exit}, L)$ where N is a set of nodes, $E \subseteq N \times T \times N$ is a set of T-labeled edges (where T is given by the target language, but must contain the label seq), $Start, Exit \in N$ are the distinguished Start and Exit nodes of the graph, and $L : N \to I$ assigns a program instruction to each node, such that: Start has no incoming edges, Exit has no outgoing edges, and the outgoing edges of each node except Exit correspond properly to the instruction label at that node, where the required correspondence is determined by the target language. A *tCFG* is a collection of disjoint CFGs, one for each thread in the program being represented. If $\mathcal{G}$ is a tCFG and t is a thread, we write $\mathcal{G}_t$ for the CFG of t in $\mathcal{G}$.

Paths through a tCFG can then be defined as sequences of vectors of program points, one per thread, and we can use CTL to state properties over tCFGs such as "no load occurs before the following store". The set of atomic predicates used in side conditions may depend on the target language under consideration; here we present some of the fairly general predicates used for our case study. These predicates break down into two types: those that depend on the state (i.e., map from threads to program points) in which they are evaluated, and those that do not (i.e., those that check some global property of the tCFG under consideration). State-based predicates include:

- $\mathsf{node}_t(n)$, which is true of a state q when $q_t = n$.
- $\mathsf{stmt}_t(i)$, which is true of a state q when the instruction at q is i in $\mathcal{G}_t$.
- $\mathsf{out}_t(ty, n')$, which is true of a state q when q_t has an outgoing edge to n' with label ty in $\mathcal{G}_t$.

State-independent predicates include:

- $\mathsf{conlit}(e)$, which is true when e represents a program constant.
- $\mathsf{varlit}(e)$, which is true when e represents a program variable (in our case study, we further distinguish between local (lvarlit) and global (gvarlit)).

Note that all of these predicates are purely syntactic static properties of tCFGs. This is not a coincidence: PTRANS optimizations can be stated and executed independently of the semantics of the target language, so that PTRANS may serve as a design tool even in the absence of formal semantics for the target language. Although we may quantify over paths in our side condition, these are paths through the *syntax* of a program as expressed in a tCFG, rather than dynamic executions of the program. Of course, when reasoning about the correctness of a transformation, we will need to relate these static properties to dynamic properties of program executions.

We also provide several extended predicates that allow the integration of outside analyses into CTL conditions. These predicates include:

- $\mathsf{cannot\_alias}_t(e, e')$, which is true of a state q when alias analysis can show that e and e' are not pointers to the same location in t at q.
- $\mathsf{in\_critical}_t(e, x)$, which is true of a state q when mutex analysis can show that q_t is part of a critical section for e protecting the value of x.
- $\mathsf{protected}_t(e, x)$, which is true when mutex analysis can show that the value of x is only changed in critical sections for e.

We incorporate these analyses by providing an axiomatization of the properties of a correct analysis (e.g., that if $\mathsf{cannot\_alias}_t(e, e')$ holds then e and e' do not point to the same location in any execution), and use these axioms to construct proofs of correctness for an optimization independently of the particular implementation of the analysis used to execute the optimization. The semantics of PTRANS actions and strategies can then be taken directly from our previous formalization of the TRANS system [11] to PTRANS. We have also developed an execution engine for PTRANS in F#, using the Z3 SMT solver to find solutions to the side conditions, so that we can test optimizations on actual CFGs before engaging in the heavy-duty work of verification.

3 Concurrent Memory Models

In order to verify optimizations on a target language, we must first provide semantics for that language – but before that, we must define our notion of concurrency. Our approach is to give operational semantics to target languages over CFGs, and to parameterize those definitions by a concurrent memory model. A concurrent memory model provides an answer to the question, "what are the values that a memory *read* operation can read?" Almost every processor architecture has its own answer to this question, and many have more than one. Adding to the confusion, many of these models, including the one specified for LLVM [9], are not *operational*; they are phrased as conditions on total executions, rather than as properties that can be checked in individual steps of an operational semantics. As part of the development of PTRANS, we have developed a general approach to specifying operational concurrent memory models. Our memory models must support four functions:

- can_read, the workhorse of the memory model, which returns the set of values that a thread can see at a given memory location
- free_set, which returns the set of locations that are free in the memory
- start_mem, which gives a default initial memory
- update_mem, which updates a memory with a set of memory operations performed by various threads

We define three instances of this axiomatization for use in our example: sequential consistency (SC), total store ordering (TSO), and partial store ordering (PSO). Sequential consistency, the most intuitive memory model, requires that every execution observed could have been produced by some total order on the memory operations in the execution. Operationally, this can be modeled by requiring each read of a location to see the most recent write to that location. We implement SC with a map from memory locations to values and a straightforward implementation of the four required functions. The function can_read looks up its target in the memory map; free_set returns the set of locations with no values in the map; start_mem is the empty map; and update_mem applies the given memory operations to the map, storing a new value on a write or arw, initializing the location with a starting value on an alloc, and clearing the location on a free.

The TSO and PSO models are slightly more complex: they allow writes to be delayed past other instructions (reads of other locations in TSO; reads and writes to other locations in PSO), resulting in executions such as the one shown (in pseudocode) in Figure 1. Under SC, if one of the read instructions returned 0 in an execution, then we would be forced to conclude that the write instruction in the same thread executed before it, and so the other read could only read a value of 1. Under TSO, however, the writes may be delayed past the

Start: $\ell_1 \mapsto 0$ and $\ell_2 \mapsto 0$

```
write $\ell_1$ 1        | write $\ell_2$ 1
$x$ := read $\ell_2$    | $y$ := read $\ell_1$
```

Result: $x = 0 \wedge y = 0$

Figure 1 Behavior forbidden by SC but allowed in TSO.

reads, allowing both reads to return 0. As shown by Sindhu et al. [14], this behavior can be modeled by associating a FIFO *write buffer* with each thread (or, for PSO, a write buffer per memory location for each thread). When a write operation is performed, it is inserted into the executing thread's write buffer; at any point, the oldest write in any thread's write buffer may be written to the shared memory. A read operation first looks for the most recent write to the location in the thread's write buffer, and if none exists reads from the location in the shared memory. In this model, atomic arw operations serve as memory fences: they are not executed until the write buffer of the executing thread is cleared.

Some optimizations, particularly those that do not involve memory in any way, may be proved correct independently of the memory model. However, one of the purposes of relaxed memory models is to allow a wider range of optimizations, so we expect that most interesting optimizations will depend on the memory model being used. In general, some memory models are strictly more permissive than others – for instance, every execution produced under SC can also be produced under TSO – but depending on our notion of correctness, it may not follow that every valid SC optimization is also a valid TSO optimization, since an SC optimization may rely on the correctness of, e.g., a locking mechanism that only functions properly under SC.

4 MiniLLVM: A Sample Intermediate Language

In this section we present MiniLLVM, a language based on the LLVM intermediate language [9], for use as a target for transformation. The syntax of MiniLLVM is defined as follows:

$$\mathit{expr} ::= \%\mathrm{x} \mid @\mathrm{x} \mid \mathrm{c} \qquad\qquad \mathit{type} ::= \mathtt{int} \mid \mathit{type}^*$$

$$\begin{aligned}\mathit{instr} ::=\ & \%\mathrm{x} = \mathrm{op}\ \mathit{type}\ \mathit{expr},\ \mathit{expr}\ | \%\mathrm{x} = \mathtt{icmp}\ \mathrm{cmp}\ \mathit{type}\ \mathit{expr},\ \mathit{expr} \mid \mathtt{br}\ \mathit{expr} \mid \mathtt{br} \mid \\ & \%\mathrm{x} = \mathtt{call}\ \mathit{type}\ (\mathit{expr},\ ...,\ \mathit{expr}) \mid \mathtt{return}\ \mathit{expr} \mid \mathtt{alloca}\ \%\mathrm{x}\ \mathit{type} \mid \\ & \%\mathrm{x} = \mathtt{load}\ \mathit{type}^*\ \mathit{expr} \mid \mathtt{store}\ \mathit{type}\ \mathit{expr},\ \mathit{type}^*\ \mathit{expr} \mid \\ & \%\mathrm{x} = \mathtt{cmpxchg}\ \mathit{type}^*\ \mathit{expr},\ \mathit{type}\ \mathit{expr},\ \mathit{type}\ \mathit{expr} \mid \mathtt{is\_pointer}\ \mathit{expr}\end{aligned}$$

(Note that the *'s indicate not repetition but pointer types.) Because the targets of control-flow instructions are implicit in the CFG, label arguments to br instructions and function names in call instructions are omitted. We give semantics to the language by specifying a labeled transition relation on program configurations. The single-thread semantics is given by the transition relation $G, t, m \vdash (p, \mathit{env}, \mathit{st}, \mathit{al}) \xrightarrow{a} (p', \mathit{env}', \mathit{st}', \mathit{al}')$ where G is the CFG representing the thread, t is the thread name, m is the shared memory, p is a program point, env is an environment giving values for thread-local variables, st is the call stack for the thread, al is a record of the memory locations allocated by the thread, and a is the set of memory operations performed by the thread. Memory operations are chosen from:

$$a ::= \quad \mathsf{read}\ t\ \mathit{loc}\ v \mid \mathsf{write}\ t\ \mathit{loc}\ v \mid \mathsf{arw}\ t\ \mathit{loc}\ v\ v \mid \mathsf{alloc}\ t\ \mathit{loc} \mid \mathsf{free}\ t\ \mathit{loc}$$

where arw represents an atomic read-and-write operation (as performed by the cmpxchg instruction). Several of the semantic rules for MiniLLVM instructions are shown in Figure 2. In the figure, Label G p indicates the instruction label assigned to node p in the CFG G, and next ℓ p indicates the node reached along an outgoing ℓ-labeled edge from p.

A concurrent configuration is a vector of configurations, one for each thread, paired with a shared memory. The concurrent semantics of MiniLLVM is given by a single rule:

$$\frac{G_t, t, m \vdash \mathit{states}_t \xrightarrow{a} (p', \mathit{env}', \mathit{st}', \mathit{al}') \qquad \mathsf{update\_mem}\ m\ a\ m'}{(\mathit{states}, m) \rightarrow (\mathit{states}(t \mapsto (p', \mathit{env}', \mathit{st}', \mathit{al}')), m')}$$

$$\frac{\mathsf{Label}\ G\ p = (\%x = \mathrm{op}\ ty\ e_1, e_2) \qquad (e_1\ op\ e_2, env) \Downarrow v}{G, t, m \vdash (p, env, st, al) \rightarrow (\mathsf{next\ seq}\ p, env(x \mapsto v), st, al)}$$

$$\frac{\mathsf{Label}\ G\ p = (\mathtt{br}\ e) \qquad (e, env) \Downarrow v \qquad v \neq 0}{G, t, m \vdash (p, env, st, al) \rightarrow (\mathsf{next\ true}\ p, env, st, al)}$$

$$\frac{\mathsf{Label}\ G\ p = (\mathtt{alloca}\ \%x\ ty) \qquad loc \in \mathsf{free\_set}\ m}{G, t, m \vdash (p, env, st, al) \xrightarrow{\mathsf{alloc}\ t\ loc} (\mathsf{next\ seq}\ p, env(x \mapsto loc), st, al \cup \{loc\})}$$

$$\frac{\mathsf{Label}\ G\ p = (\mathtt{store}\ ty_1\ e_1, {ty_2}^*\ e_2) \qquad (e_1, env) \Downarrow v \qquad (e_2, env) \Downarrow loc}{G, t, m \vdash (p, env, st, al) \xrightarrow{\mathsf{write}\ t\ loc\ v} (\mathsf{next\ seq}\ p, env, st, al)}$$

Figure 2 Some single-thread transition rules for MiniLLVM.

In other words, we produce a concurrent step simply by selecting one thread to take a step, and then updating the memory with the memory operations performed by that thread.

5 Verification

5.1 Defining Correctness

Before we can begin verifying an optimization, we must clearly state what it means for an optimization to be correct. The semantics of a compiler transformation can be expressed denotationally in terms of the program graphs that may be produced as a result of the transformation on a given input graph. We can call a transformation T correct if, for any graph G, any graph G' output by applying T to G has some desired property relative to G. We will use *observational refinement* [3] as our sense of correctness; in other words, we will require that any observable behavior of G' is also an observable behavior of G, implying that T does not introduce any new behaviors. We will prove this refinement via *simulation* [2]:

▶ **Definition 2.** A *simulation* is a relation $\preceq$ on two labeled transition systems P and Q such that for any states p, p' of P and q of Q, for any label k, if $p \preceq q$ and $p \xrightarrow{k}_P p'$, then $\exists q'.\ q \xrightarrow{k}_Q q'$ and $p' \preceq q'$. By abuse of notation we write $P \preceq Q$ and say that Q simulates P.

The concurrent step relation of MiniLLVM as presented is unlabeled, but we can add labels to indicate the portion of the program's behavior that should be considered observable, which will generally be some portion of the shared memory. For each optimization to be verified, we will choose the maximum possible subset of shared memory as our observables, and state a simulation relation that relates any transformed graph to its original input. (Note that for more complex optimizations, more flexible relations such as weak (stuttering) simulation may be required, but the overall structure of the proof will remain unchanged.)

While PTRANS is expressive enough to allow optimizations that transform multiple threads simultaneously, many optimizations (especially concurrent retoolings of sequential optimizations) only transform a single thread. The following theorem allows us to extend a correct simulation relation on states in a single-thread CFG to one on entire tCFG states:

▶ **Definition 3.** Let the execution state of a multithreaded program with tCFG $\mathcal{G}$ be a pair $(states, m)$, where $states$ is a vector of per-thread execution states and m is a shared memory.

The *lifting* of a simulation relation $\preceq$ on single-threaded CFGs to concurrent execution states relative to a thread t is defined by $(states, m)\ \lceil\preceq\rceil_t\ (states', m') \triangleq (states_t, m) \preceq (states'_t, m') \wedge \forall u \neq t.\ states_u = states'_u$.

▶ **Theorem 4.** *Fix a memory model supporting the functions* free_set*,* can_read*, and* update_mem*. Let $\mathcal{G}$ be a tCFG, t be a thread in $\mathcal{G}$, and obs be the set of observable memory locations. Suppose that $\preceq$ is a simulation relation such that $\mathcal{G}'_t \preceq \mathcal{G}_t$, $\mathcal{G}'_u = \mathcal{G}_u$ for all $u \neq t$, and for all $(s', m') \preceq (s, m)$ the following hold:*

1. free_set m = free_set m'
2. *For any $u \neq t$, if $u, \mathcal{G}_u, m' \vdash s_1 \xrightarrow{a} s_2$, then* can_read $m\ u\ \ell$ = can_read $m'\ u\ \ell$ *for every location ℓ mentioned in an operation in a*
3. *For any $u \neq t$, if $u, \mathcal{G}_u, m \vdash s_1 \xrightarrow{a} s_2$ and* update_mem $m'\ a\ m'_2$ *holds, then there exists some m_2 such that* update_mem $m\ a\ m_2$ *holds, $m_2|_{obs} = m'_2|_{obs}$, and $(s', m'_2) \preceq (s, m_2)$*

Then $\lceil\preceq\rceil_t$ is a simulation relation such that $\mathcal{G}'\ \lceil\preceq\rceil_t\, \mathcal{G}$.

While the exact conditions of the theorem are complicated, the intuition is straightforward: if $\preceq$ is a simulation relation for $\mathcal{G}_t$ and $\mathcal{G}'_t$ such that $(s', m') \preceq (s, m)$ implies that m and m' look the same to all threads $u \neq t$, and $\preceq$ is preserved by steps of threads other than t, then $\lceil\preceq\rceil_t$ is a simulation relation for $\mathcal{G}$ and $\mathcal{G}'$. This theorem allows us to break proofs of correctness for transformations on multithreaded programs into two parts: correctness of the simulation on the transformed thread, and validity of the relation with respect to the remaining threads. Note that in the case in which the simulation relation requires that $m = m'$, i.e., in which the optimization does not change the effects of $\mathcal{G}_t$ on shared memory, most of these conditions are trivial. In optimizations that affect the shared memory, on the other hand, the proof of the theorem's premises will involve some effort.

5.2 Specifying an Optimization

In the following sections, we will show the use of PTRANS in verifying an optimization. The candidate optimization is Redundant Store Elimination (RSE), which eliminates stores that are always overwritten before they are used, as in Figure 3. Note that s is replaced by an `is_pointer` instruction, rather than being eliminated entirely, to preserve failures: if e_2 is not pointer-valued at s the program will fail immediately, while eliminating s would allow the program to run until reaching s', potentially introducing new behavior.

In sequential code, the optimization is safe if, between the eliminated store s and the following store s', the location referred to by e_2 is not read and the value of e_2 is not changed. In the concurrent case, the correctness condition is more complex, since changes to a memory location can be observed by other threads. We will give a correct version of RSE for each of our three memory models. We begin with the rewrite portion of the transformation, which is the same in all cases, and the common portion of the side condition: the basic pattern that

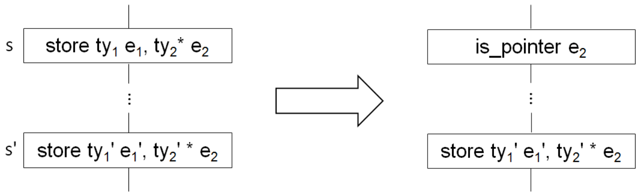

Figure 3 Redundant Store Elimination.

describes the node to be transformed, and a placeholder for the remaining conditions (note that the condition is checked starting at the entry node of the tCFG):

$$\begin{array}{l}\mathsf{replace}\ n\ \mathsf{with}\ \texttt{is\_pointer}\ e_2\ \text{if}\\ EF\ \mathsf{node}_t(n) \land \mathsf{stmt}_t(\texttt{store}\ ty_1\ e_1, ty_2^*\ e_2) \land \varphi\end{array}$$

Now, for each memory model, we need only provide a condition φ that ensures that the optimization is safe to perform. In general, this will be an "until"-property stating necessary conditions on the nodes between n and the next store to e_2.

Sequential consistency, the most restrictive of our three memory models, naturally has the most restrictive side condition. There are two approaches to securing the optimization: we could require that no memory operations occur between n and the following store, or we could require that e_2 be private to t. In this example we will take the second approach, using an external mutual exclusion analysis to ensure that e_2 is not exposed to other threads while t is in the region between n and the following store to e_2. Using the mutex predicates described in Section 2.2 and a defined $\mathsf{not\_touches}_t$ predicate that checks that a given memory location cannot be read or modified by t, the condition can be written as:

$$\begin{array}{l}\varphi_{SC} \triangleq \mathsf{protected}(x, e_2) \land \mathsf{gvarlit}(e_2) \land \neg\mathsf{is}(x, e_2)\ \land\\ \quad A\ \mathsf{in\_critical}_t(x, e_2) \land (\mathsf{node}_t(n) \lor \mathsf{not\_touches}_t(e_2))\\ \quad \mathcal{U}\ (\mathsf{in\_critical}_t(x, e_2) \land \neg\mathsf{node}_t(n)\ \land \exists ty_1', e_1', ty_2'.\ \mathsf{stmt}_t(\texttt{store}\ ty_1' e_1', ty_2'\ e_2))\end{array}$$

Next we will consider the appropriate side condition for the TSO memory model. Since TSO allows writes to be delayed past certain other operations, in a program with a redundant store, it is possible that the redundant store may be delayed until immediately before the following store to e_2. If this behavior is possible in the original program, then removing the store will not introduce new behavior. Thus, our side condition need only characterize the circumstances under which the store at n could have been delayed in the original program. In TSO, a write can be delayed past reads to different locations, but not past writes or atomic read-writes. Thus, the necessary side condition is as follows, where $\mathsf{not\_loads}$ checks that no `load` instructions read from a location and $\mathsf{not\_mods}$ ensured that the value of an expression is not changed:

$$\begin{array}{l}\varphi_{TSO} \triangleq AX_t(A\ \mathsf{not\_mods}_t(e_2) \land \mathsf{not\_loads}_t(e_2)\ \land\\ \quad \neg(\exists x, ty_1, e_1, ty_2, e_2', ty_3, e_3.\ \mathsf{stmt}_t(\texttt{store}\ ty_1\ e_1, ty_2^*\ e_2') \lor\\ \quad \mathsf{stmt}_t(\%x = \texttt{cmpxchg}\ ty_1^*\ e_1, ty_2\ e_2', ty_3\ e_3))\\ \quad \mathcal{U}\ (\neg\mathsf{node}_t(n)\ \land \exists ty_1', e_1', ty_2'.\ \mathsf{stmt}_t(\texttt{store}\ ty_1'\ e_1', ty_2'\ e_2)))\end{array}$$

where AX_t is a derived temporal operator defined such that $AX_t\varphi$ iff φ is true in every state in which the thread t has advanced by one node (regardless of the behavior of other threads). The fragment of the condition inside the AX_t operator provides a useful characterization of the nodes between n and the following store to e_2; we will call it φ'_{TSO}, where $\varphi_{TSO} = AX_t\ \varphi'_{TSO}$. Note that φ_{SC} is also a reasonable side condition under TSO, and we could form a more general optimization by using $\varphi_{SC} \lor \varphi_{TSO}$ as our side condition.

The relaxation of the PSO memory model is a more permissive version of that of TSO, so we can obtain a side condition for it by relaxing the constraints of φ_{TSO}. A write in PSO can be delayed past reads and writes to different locations, but not past operations on the same location or atomic read-writes, so the corresponding side condition is:

$$\begin{array}{l}\varphi_{PSO} \triangleq AX_t(A\ \mathsf{not\_mods}_t(e_2) \land \mathsf{not\_touches}_t(e_2)\ \land\\ \quad \neg(\exists x, ty_1, e_1, ty_2, e_2', ty_3, e_3.\ \mathsf{stmt}_t(\%x = \texttt{cmpxchg}\ ty_1^*\ e_1, ty_2\ e_2', ty_3\ e_3))\\ \quad \mathcal{U}\ (\neg\mathsf{node}_t(n)\ \land \exists ty_1', e_1', ty_2'.\ \mathsf{stmt}_t(\texttt{store}\ ty_1' e_1', ty_2'\ e_2)))\end{array}$$

This condition is strictly weaker than φ_{TSO}, allowing the optimization to be applied to a wider range of programs. As above, we also define φ'_{PSO} such that $\varphi_{PSO} = AX_t\ \varphi'_{PSO}$ for use in our proofs of correctness.

5.3 Verification of RSE

We are now ready to demonstrate the correctness of MiniLLVM RSE in PTRANS. As laid out in Section 5.1, we prove correctness by showing that for any transformed tCFG $\mathcal{G}'$ produced by applying the optimization to a graph $\mathcal{G}$, there exists a simulation relation $\preceq$ such that $\mathcal{G}'_t \preceq \mathcal{G}_t$, states related by $\preceq$ make the same values visible to threads other than t, and steps by threads other than t preserve $\preceq$. For each version of RSE, we will present such a relation and sketch the proof of its correctness.

▶ **Theorem 5.** *Let $\mathcal{G}'$ be a tCFG in the output of RSE(φ_{SC}) on a tCFG $\mathcal{G}$, and ℓ be the location targeted by the redundant store removed in $\mathcal{G}'$. Let $\preceq_{SC}$ be the relation such that $(s', m') \preceq_{SC} (s, m)$ iff*

- *$s = s'$*
- *either $\ell \in \mathsf{free\_set}\ m$ and $\ell \in \mathsf{free\_set}\ m'$, or $\ell \notin \mathsf{free\_set}\ m$ and $\ell \notin \mathsf{free\_set}\ m'$*
- *either $m = m'$, or else φ_{SC} holds at the program point of s in $\mathcal{G}$ and $m|_{\overline{\ell}} = m'|_{\overline{\ell}}$.*

Then $\lceil \preceq_{SC} \rceil_t$ is a simulation relation such that $\mathcal{G}'\ \lceil \preceq_{SC} \rceil_t\ \mathcal{G}$ with all locations other than ℓ observable.

Proof. Consider two related states (s, m) of $\mathcal{G}_t$ and (s', m') of $\mathcal{G}'_t$. In case (1), the only interesting case is the one in which s is at the transformed node n; in this case, $\mathcal{G}'_t$ executes the `is_pointer` instruction and $\mathcal{G}_t$ executes the `store` instruction. Since the side condition of the RSE transformation is true on $\mathcal{G}$, we know that φ_{SC} holds at n, and so $\preceq_{SC}$ holds on the resulting states. If, on the other hand, we are in case (2), then we know that φ_{SC} holds, so s must be in the region between n and the next store to e_2. If we have not yet reached the next store to e_2, then since $\preceq_{SC}$ holds we know that it does not read or modify the memory at ℓ, and we can conclude that $\mathcal{G}_t$ and $\mathcal{G}'_t$ execute the same instruction and arrive in new configurations (s_2, m_2) and (s'_2, m'_2) such that m_2 and m'_2 differ only at ℓ and φ_{SC} still holds. The guarantees of mutual exclusion ensure the separation of threads required by Theorem 4, and we can conclude that $\lceil \preceq_{SC} \rceil_t$ is a simulation relation showing the correctness of the SC version of RSE. ◀

Recall that, while in SC the memory is simply a map m from locations to values, in TSO and PSO it is a pair (m, b) of a shared memory and per-thread write buffers. Since the correctness of our conditions under these models depends on our ability to delay stores until they become redundant, we must have a notion of one buffer being a "redundant expansion" of another.

▶ **Definition 6.** A *write buffer* is a queue of writes expressed as location-value pairs. A write buffer b' is a *redundant expansion* of b if b' can be constructed from b by adding, in front of each pair (ℓ, v) in b, zero or more writes of other values to ℓ. We will say that a collection of write buffers c' is a redundant expansion of a collection c when each write buffer c'_t is a redundant expansion of the corresponding write buffer c_t.

Because the added writes appear immediately in front of other writes to the same location, they can be immediately overwritten when the buffers are cleared, and are never read when looking for the latest write to a location. This allows a redundant expansion of b to simulate the behavior of b with regard to the memory-model functions.

▶ **Theorem 7.** *Let $\mathcal{G}'$ be a tCFG in the output of RSE(φ_{TSO}) on a tCFG $\mathcal{G}$. Let $\preceq_{TSO}$ be the relation such that $(s', (m', b')) \preceq_{TSO} (s, (m, b))$ iff*

- *$s = s'$, $m = m'$, and $b_u = b'_u$ for all $u \neq t$, and*
- *either (1) b_t is a redundant expansion of b'_t, or else (2) φ'_{TSO} holds at the program point of s in $\mathcal{G}$, the store eliminated in $\mathcal{G}'$ was to some expression e_2, and there is a location ℓ such that e_2 evaluates to ℓ in s, the last write in b_t is a write to ℓ, and the rest of b_t is a redundant expansion of b'_t.*

Then $\lceil \preceq_{TSO} \rceil_t$ is a simulation relation such that $\mathcal{G}' \lceil \preceq_{TSO} \rceil_t \mathcal{G}$ with all locations observable.

Proof. By Theorem 4. Consider two related states $(s, (m, b))$ of $\mathcal{G}_t$ and $(s', (m', b'))$ of $\mathcal{G}'_t$. If b_t is a redundant expansion of b'_t (case 1), then the only interesting case is the one in which s is at the transformed node n; in this case, $\mathcal{G}'_t$ executes the `is_pointer` instruction, and $\mathcal{G}_t$ executes the `store` instruction, evaluating e_2 to some location ℓ and adding a write to ℓ to its buffer – thus the resulting buffer has the structure described in case (2). Since the side condition of the RSE transformation is true on $\mathcal{G}$, we know that $\varphi_{TSO} = AX_t\ \varphi'_{TSO}$ holds at n, and so $\preceq_{TSO}$ holds on the resulting states. If, on the other hand, we are in case (2), s must be in the region between n and the next store to e_2. If s is at a store to e_2 other than n, then both $\mathcal{G}$ and $\mathcal{G}'$ commit a write to ℓ; since b_t was a redundant expansion of b'_t followed by a write to ℓ, this new write makes the last one redundant, and we are now in case (1). If s is somewhere between n and the following store, then since φ'_{TSO} holds we know that the current instruction does not read the memory at ℓ and is neither a `store` nor a `cmpxchg`, so we can conclude that $\mathcal{G}_t$ and $\mathcal{G}'_t$ execute the same instruction with the same result, that the instruction adds no new writes to t's write buffer, and that the extra write to ℓ in b_t is not forced into main memory (as it would be by a `cmpxchg` instruction). Thus, case (2) of $\preceq_{TSO}$ still holds. Since the only difference in states allowed by $\preceq_{TSO}$ is in the write buffer for t, which is neither visible to nor affected by threads other than t, the separation of threads required by Theorem 4 holds, and we can conclude that $\lceil \preceq_{TSO} \rceil_t$ is a simulation relation showing the correctness of the TSO version of RSE. ◀

▶ **Theorem 8.** *Let $\mathcal{G}'$ be a tCFG in the output of RSE(φ_{PSO}) on a tCFG $\mathcal{G}$. Let $\preceq_{PSO}$ be the relation such that $(s', (m', b')) \preceq_{PSO} (s, (m, b))$ iff*

- *$s = s'$, $m = m'$, $b_{u,\ell} = b'_{u,\ell}$ for all ℓ and all $u \neq t$, and*
- *either (1) $b_{t,\ell}$ is a redundant expansion of $b'_{t,\ell}$ for all ℓ, or else (2) φ'_{PSO} holds at the program point of s in $\mathcal{G}$, the store eliminated in $\mathcal{G}'$ was to some expression e_2, and there is a location ℓ such that e_2 evaluates to ℓ in s, $b_{t,\ell}$ is a redundant expansion of $b'_{t,\ell}$ followed by a write to ℓ, and $b_{t,\ell'}$ is a redundant expansion of $b'_{t,\ell'}$ for all other locations ℓ'.*

Then $\lceil \preceq_{PSO} \rceil_t$ is a simulation relation such that $\mathcal{G}' \lceil \preceq_{PSO} \rceil_t \mathcal{G}$ with all locations observable.

Proof. By Theorem 4. The proof is nearly identical to that of the TSO case. Since write buffers are per-location, `store` instructions to locations other than ℓ may be executed between the eliminated store and the following write to ℓ without changing the relationship between $b_{t,\ell}$ and $b'_{t,\ell}$, justifying the weaker side condition; otherwise, the proof proceeds entirely analogously. ◀

In this manner, PTRANS allows us to express and verify optimizations under a variety of memory models, sharing information between specifications and proofs of similar transformations. All of the above proofs have been carried out in full formal detail in the Isabelle proof assistant, and can be found online at `http://web.engr.illinois.edu/~mansky1/ptrans`.

6 Conclusions and Related Work

In this paper we present the PTRANS specification language, in which optimizations are expressed as conditional rewrites on program syntax, and show how it can be used to state and verify compiler optimizations. We outline a method for stating and verifying optimizations that transform a single thread in a multithreaded program, with some parts independent of and others dependent on the memory model under consideration. We use this method to verify a redundant store elimination optimization on an LLVM-based language under three memory models, showing that the behaviors of every output program are possible behaviors of the input program. In combination with the executable semantics for PTRANS, which allows PTRANS specifications to serve as prototype optimizations [10], the methodology here presented forms the basis of a new framework for specifying, testing, and verifying compiler optimizations in the presence of concurrency.

Our work builds on the TRANS approach due to Kalvala et al. [4]. Among the tools that build on this approach is the Cobalt specification system [6], which aims to automatically prove the correctness of optimizations. This automation comes at the cost of expressiveness: Cobalt is limited to a much smaller set of CTL side conditions than TRANS (or PTRANS) in general. While interactive proofs require considerably more effort, using a standard framework for proofs across different memory models and target languages can reduce the burden by allowing common facts (about simulation, CTL over CFGs, etc.) to be proved once and for all. To the best of our knowledge, neither Cobalt nor any other TRANS-style work has yet addressed the problem of concurrency.

The most comprehensive compiler correctness effort to date is CompCertTSO [16], the extension of CompCert [7] to the TSO memory model. CompCertTSO includes a range of verified optimizations on intermediate languages at various levels, as well as verified translations between languages, while we have thus far only verified same-language transformations. Our approach has the advantage of language- and memory-model independence; our framework also allows us to separate out the correctness condition for a concurrent optimization into a simulation relation on a single thread and side conditions on the remaining threads, while the one concurrency-aware optimization verified in CompCertTSO involves a whole-program simulation proof. Ševčík [15] has also verified various optimizations, including redundant instruction eliminations, in a language-independent manner under data-race-free sequential consistency, specifying optimizations directly as transformations on the executions traces of programs (which may not directly correspond to modification of program code).

Burckhardt et al. [1] have developed a method of verifying optimizations under relaxed memory models by specifying memory models as sets of rewrite rules on program traces and optimizations as rewrites on local fragments of a program. Their proofs are fully automatic, using the Z3 SMT solver to check that all traces allowed by a transformed program fragment could be produced by applying the rewrite rules allowed by the memory model to the traces of the original program. They rely on a denotational semantics for their target language that gives the set of possible program traces for every program, and thus far have only verified transformations on single instructions or pairs of immediately adjacent instructions (including a simple RSE); their method does not obviously extend to transformations that require analysis over fragments of the program graph of indefinite size (e.g., all the instructions between one instruction and another).

Thus far, we have only verified optimizations that transform one thread at a time, assisted by a theorem that allows us to lift single-thread simulation relations to simulations on multithreaded CFGs. If we expand our scope to optimizations that transform multiple threads simultaneously (as might be done in some lock-related transformations), we may require both an extended language of side conditions and more general proof techniques, such as the rely-guarantee approach found in RGSim [8]. Similar approaches may help us handle other models of parallel programming, such as fork-join parallelism.

References

1 Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. Verifying local transformations on relaxed memory models. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 104–123, Berlin, Heidelberg, 2010. Springer-Verlag.

2 Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer Berlin / Heidelberg, 1980. doi: 10.1007/3-540-10003-2_79.

3 C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

4 Sara Kalvala, Richard Warburton, and David Lacey. Program transformations using temporal logic side conditions. *ACM Trans. Program. Lang. Syst.*, 31(4):1–48, 2009.

5 Jens Krinke. Context-sensitive slicing of concurrent programs. *SIGSOFT Softw. Eng. Notes*, 28(5):178–187, September 2003.

6 Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. *SIGPLAN Not.*, 38:220–231, May 2003.

7 Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.

8 Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'12, pages 455–468, New York, NY, USA, 2012. ACM.

9 LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html, April 2014.

10 William Mansky, Dennis Griffith, and Elsa L. Gunter. Specifying and executing optimizations for parallel programs. Accepted for publication by GRAPHITE'14.

11 William Mansky and Elsa Gunter. A framework for formal verification of compiler optimizations. In *Proceedings of the First international conference on Interactive Theorem Proving*, ITP'10, pages 371–386, Berlin, Heidelberg, 2010. Springer-Verlag.

12 Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

13 Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS'98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166, London, UK, 1998. Springer-Verlag.

14 Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. In Michel Dubois and Shreekant Thakkar, editors, *Scalable Shared Memory Multiprocessors*, pages 25–41. Springer US, 1992.

15 Jaroslav Ševčík. Safe optimisations for shared-memory concurrent programs. *SIGPLAN Not.*, 46(6):306–316, June 2011.

16 Jaroslav Ševčik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. *SIGPLAN Not.*, 46(1):43–54, January 2011.

17 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.