A Cross-Language Framework for Verifying Compiler Optimizations

William Mansky, Elsa L. Gunter

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

Abstract. Most compiler correctness efforts, whether based on validation or once-and-for-all verification, are tightly tied to the particular language(s) under consideration. Proof techniques may be replicated for other targets, and parts of the compiler chain may be shared for new input or output languages, but the extent to which common elements can be generalized across multiple targets has not been fully explored. In this paper, we lay out a general approach to specifying and verifying optimizations and transformations on low-level intermediate languages. By generalizing across elements such as concurrent memory models and single-thread operational semantics, we can build a library of facts that can be reused in verifying optimizations for dramatically different target languages, such as stack-machine and register-machine languages.

Keywords: optimizing compilers, parallel programming, interactive theorem proving, program transformations, temporal logic

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 [23]. 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 [17]), or by general proof of correctness for all possible inputs (the approach taken, for instance, in the CompCert verified C compiler [10]). As Morisset et al. [16] have shown, the problem is only aggravated when dealing with optimization of parallel programs, a rapidly developing field in which many algorithms are still poorly understood. Insufficiently analyzed optimizations may result in unreliable execution of parallel 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 parallel optimizations.

CompCert has demonstrated that it is possible to verify every phase of a compiler once and for all. However, CompCert is ultimately only one compiler, albeit with several different source and target languages. The goal of the VeriF-OPT project is to develop basic theories and methodologies that will facilitate the development of new CompCert-like projects, by providing a language-independent framework for specifying and verifying compiler optimizations. In this paper, we present a domain-specific, formal, and target-independent language for specifying compiler optimizations, and demonstrate its use in verifying an optimization across multiple targets and memory models. We will see that the use of a unified framework for multiple target languages allows us to generalize much of the proof effort and reuse proof components, simplifying the process of verifying each individual optimization.

2 The PTRANS Specification Language

2.1 PTRANS: Adapting TRANS to Parallel Programs

The core of our approach is the PTRANS specification language, an extension of the TRANS language [5] designed with concurrency-awareness and language-independence as first principles. The basic approach is drawn from the work of Lacey et al. [7]: the transformation to be made is specified as a rewrite on program code in the form of a control flow graph (CFG), and the conditions under which the optimization may be applied are expressed in terms of temporal logic. Our starting point is our previous formalization of the syntax and semantics of TRANS for sequential programs [13]. The syntax of PTRANS is given by the following grammar:

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

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. 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. Kalvala et al. have shown that a wide variety of common program transformations can be expressed using these basic rewrites

At the top level, a transformation T is built out of conditional rewrites combined with *strategies*. The expression $A_1, ..., A_m$ if φ is the basic pairing of one or more rewrites with a CTL side condition. MATCH φ IN T provides an additional side condition for a set of transformations, and allows metavariables to be bound across multiple rewrites. The THEN and $\square$ operators provide sequencing and choice respectively, and APPLY_ALL T recursively applies T wherever possible until it is no longer applicable to the graph under consideration.

The primary aim of the PTRANS framework is to serve as a language-independent platform for the verification of new compiler optimizations and transformations. Wherever possible, its definition is *parameterized* by the details of the target language. Rather than incorporate language structure or expected transformation patterns into the framework, we leave such elements unspecified except for a minimal set of axioms. We have already seen one parameter provided by the target language: the language of instruction labels with metavariables. In Section 3 we will introduce two sample instantiations of the language parameter, which themselves take memory models as parameters, and in Section 6 we will verify optimizations within the context of these instantiations.

The TRANS 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 parallel analogue to the CFG in order to extend the approach to parallel programs. The model used here, adapted from the work of Krinke [6], is the threaded control flow graph (tCFG): a collection of non-intersecting CFGs, one for each thread in a program. Formally:

Definition 1. *A* CFG *is a labeled directed graph* $(N, E, Start, Exit, L)$ *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* $\mathcal{G}$*.*

The revised semantics of TRANS rewrites [13] can be straightforwardly extended to tCFGs: since individual CFGs do not intersect, the nodes mentioned in each atomic action uniquely identify (at most) one thread on which to perform the rewrite. If nodes from several different threads are mentioned in a single atomic action, the rewrite fails to apply, as in the case where the nodes mentioned do not exist in the graph.

The set of atomic predicates used in side conditions may depend on the target language under consideration; we give a set of basic atomic predicates that are likely to be useful in any target language, and allow languages to provide their own additional predicates as well. Atomic predicates break down into two types: those that depend on the state (i.e., the vector of program points in a tCFG) in which they are evaluated, and 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:

- $\mathsf{is}(x, y)$, which is true when the metavariables x and y represent the same program object (number, node, instruction, etc.).
- $\mathsf{int\_eq}(a, b)$, which is true when a and b are arithmetic expressions that statically evaluate to the same value (according to some reference semantics for arithmetic operations).

Note that all of these predicates are purely syntactic static properties of tCFGs. In general, PTRANS optimizations can be stated and performed 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. Of course, semantics will be required to reason about the correctness of optimizations. We may also want to rely on the results of external analyses whose correctness depends on the program semantics; we will explore this idea in more detail in Section 5.

3 Intermediate Languages for PTRANS

Control flow graphs are most commonly used as a code representation for compiler intermediate languages, and particularly for those in which each instruction is in some sense a single unit of computation. In this section, we will describe two such languages used as targets for PTRANS optimization. The first, MiniLLVM, is a register-machine language, in which an instruction consists of an operator together with several operands (constants, variables, or simple expressions) and intermediate results are stored in local registers. The second, GraphBIL, takes the contrasting stack-machine approach, in which intermediate results are stored in and operands drawn from a stack of computed values. For each language, we will present the syntax of CFG node labels (i.e., instructions) for the language and give a small-step transition semantics that shows how each node is executed.

3.1 MiniLLVM

Our first target language is MiniLLVM, a language based on the LLVM intermediate representation [8]. We omit labels from our instructions, since the targets of jumps can be determined from the control flow graph. We also omit several formal details of the IR: for instance, while types are included in the syntax, we perform very little dynamic type checking, and while the LLVM IR is always in static single assignment form, MiniLLVM programs are not in SSA by default. This simplifies the formalization and reduces the number of well-formedness constraints that need to be carried through proofs of correctness. The syntax of MiniLLVM instructions is defined as follows:

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

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

The one instruction not present in the LLVM IR is `is_pointer`, which checks whether a given expression is pointer-valued but otherwise does nothing; we will use this as a replacement for eliminated memory operations in our optimization. (Note that the *'s indicate not repetition but pointer types.)

A CFG structure labeled with MiniLLVM instructions is only a well-formed CFG if a certain relationship holds between the label on each node and the outgoing edges of that node. This relationship is defined as follows:

- A node labeled with a conditional branch `br` e, has one outgoing edge labeled true and one labeled false.
- A node labeled with function call `call` ty $(e_1, ..., e_n)$ has one outgoing edge labeled call (leading to the function body) and one labeled seq (indicating the location to which to return once the call is complete).
- A node labeled with a return instruction `return` e can have any number of outgoing edges, all labeled ret.
- A node with any other label has one outgoing edge labeled seq.

We give semantics to the language by specifying a labeled transition relation on program configurations. The semantics of an individual thread are given by the transition relation $t, G, m \vdash (n, \mathit{env}, \mathit{st}, \mathit{al}) \xrightarrow{a} (n', \mathit{env}', \mathit{st}', \mathit{al}')$ where G is the CFG representing the thread, t is the thread name, m is the shared memory, n is a node in G, 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:

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

where arw represents an atomic read-and-write operation (as performed by the `cmpxchg` instruction). The semantics are parameterized by a *memory model* that provides functions can_read, free_set, and update_mem for interacting with the shared memory (as explained in detail in Section 4). Several of the semantic rules for MiniLLVM instructions are shown in Figure 1. In the figure, Label G n indicates the instruction label assigned to node n in the CFG G, and next ℓ n indicates the node reached along an outgoing ℓ-labeled edge from n. We proceed through the graph G by looking up the instruction label at the current node, performing some transformation on the thread-local state and producing some memory operations, and moving on to another node as indicated by the edges of G.

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

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

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

$$\frac{\mathsf{Label}\ G\ n = (\texttt{is\_pointer}\ e) \qquad n \neq \mathsf{Exit}\ G \qquad (e, \mathit{env}) \Downarrow \mathit{loc}}{t, G, m \vdash (n, \mathit{env}, \mathit{st}, \mathit{al}) \rightarrow (\mathsf{next\ seq}\ n, \mathit{env}, \mathit{st}, \mathit{al})}$$

Fig. 1. Some single-thread transition rules for MiniLLVM

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

$$\frac{t, \mathcal{G}_t, m \vdash \mathit{states}_t \xrightarrow{a} (n', \mathit{env}', \mathit{st}', \mathit{al}') \qquad \mathsf{update\_mem}\ m\ a\ m'}{(\mathit{states}, m) \to (\mathit{states}(t \mapsto (n', \mathit{env}', \mathit{st}', \mathit{al}')), m')}$$

In other words, we produce a concurrent step simply by selecting one thread in the tCFG $\mathcal{G}$ to take a step, and then updating the memory with the memory operation (if any) performed by that thread. As we will see in the next section, once we have single-thread CFG semantics for a language, a rule of this form can be given to produce concurrent semantics without any specific reference to the language under consideration.

3.2 GraphBIL

Our second target language, GraphBIL, is adapted from Gordon and Syme's Baby IL (BIL) [2]. BIL is itself a simple subset of the Common Intermediate Language (CIL) [1], which is used as an intermediate representation for the compilers of the .NET Framework. As before, we define a set of instructions for use as node labels on CFGs; except for the language of instructions and the relationship between node labels and outgoing edges, the CFGs of GraphBIL are identical to those of MiniLLVM. The syntax of GraphBIL instructions is defined as follows:

$$\begin{aligned}\mathit{instr} ::= {} & \texttt{ldc.i4}\ \mathit{int} \mid \texttt{br} \mid \texttt{brtrue} \mid \texttt{brfalse} \mid \texttt{ldind} \mid \texttt{stind} \mid \texttt{ldarga}\ \mathit{int} \mid \texttt{starg}\ \mathit{int} \mid \\ & \texttt{newobj void}\ \mathit{class}{::}\,\texttt{.ctor}(\mathit{type}, ..., \mathit{type}) \mid \texttt{callvirt}\ \mathit{type}\ \mathit{class}{::}\mathit{method}(\mathit{type}, ..., \mathit{type}) \mid \\ & \texttt{call instance}\ \mathit{type}\ \mathit{class}{::}\mathit{method}(\mathit{type}, ..., \mathit{type}) \mid \texttt{ret} \mid \texttt{ldflda}\ \mathit{type}\ \mathit{class}{::}\mathit{field} \mid \\ & \texttt{box}\ \mathit{class} \mid \texttt{unbox}\ \mathit{class} \mid \texttt{dup} \mid \texttt{pop} \mid \texttt{pop\_pointer}\end{aligned}$$

Most of the instructions are straightforwardly derived from BIL, with arguments drawn from the evaluation stack rather than explicit in the instructions; we provide several lower-level instructions from CIL, such as `br` and `ret`, to reflect our lower-level program model. We also add three instructions that are useful in optimizing stack machine programs: `dup` duplicates the top element of the evaluation stack, `pop` removes the top element, and `pop_pointer` emulates the behavior of a `stind` instruction without actually performing a store (i.e., it removes the top two values from the stack after checking that the second is a pointer).

The allowed outgoing edge types for each GraphBIL instruction are as follows:

- A node labeled with a conditional branch `brtrue` or `brfalse` has one outgoing edge labeled true and one labeled false.
- A node that calls a method of an unboxed object `call instance` $(B\ vc :: \ell(A_1, ..., A_n))$ has one outgoing edge labeled call vc (leading to the method body) and one labeled seq (indicating the location to which to return once the call is complete).
- A node that calls a method of a boxed object `callvirt` $(B\ c :: \ell(A_1, ..., A_n))$ has one outgoing edge labeled seq (indicating the location to which to return once the call is complete), and one edge labeled mcall c' for each class c' such that $c' <: c$.
- A node labeled with a return instruction `return` e can have any number of outgoing edges, all labeled seq.
- A node with any other label has one outgoing edge labeled seq.

Of particular interest is the requirement for the `callvirt` instruction, which ensures that a virtual method call can be dynamically dispatched to any subclass of the class given in the instruction.

The semantics of GraphBIL are derived systematically from those of BIL, making several elements of the CIL execution state explicit: in particular, the evaluation stack (on which operands to future instructions are placed), the program point, and the distinction between thread-local state and shared memory. We also flatten out the structure of the heap: while in BIL field references can be attached to either stack or heap pointers, in GraphBIL we resolve field references in the heap to unstructured locations. For simplicity, we restrict the `stind` instruction to store only to heap addresses, and use `starg` to store to the argument vectors

$$\frac{\mathsf{Label}\ G\ n = \texttt{brtrue} \quad n \neq \mathsf{Exit}\ G \quad e = \text{if } v = 0 \text{ then } \mathsf{seq} \text{ else } \mathsf{branch}}{t, G, h \vdash (s, vs, args, n) \rightarrow (s, vs, args, \mathsf{next}\ G\ e\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{stind} \quad n \neq \mathsf{Exit}\ G}{t, G, h \vdash (s, vs; p; v, args, n) \xrightarrow{\mathsf{write}\ t\ p\ v} (s, vs, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\begin{array}{c}\mathsf{Label}\ G\ n = \texttt{starg}\ j \quad n \neq \mathsf{Exit}\ G \quad s = s_1\ \ldots\ s_i \\ \mathsf{stack\_update}(s, args, (i+1, j), v) = (s, args')\end{array}}{t, G, h \vdash (s, vs; v, args, n) \rightarrow (s, vs, args', \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\begin{array}{c}\mathsf{Label}\ G\ n = \texttt{newobj void}\ c{::}\ \texttt{.ctor}(A_1, \ldots, A_k) \quad n \neq \mathsf{Exit}\ G \quad c \notin \mathsf{ValueClass} \quad p \in \mathsf{free\_set}\ h \\ \mathsf{fields}(c) = f_i \mapsto {A_i}^{i \in 1..n}\end{array}}{t, G, h \vdash (s, vs; v_1; \ldots; v_k, args, n) \xrightarrow{\mathsf{write}\ t\ p\ c[f_i \mapsto {v_i}^{i \in 1..n}]} (s, vs; p, args, \mathsf{next}\ G\ \mathsf{seq}\ n)}$$

$$\frac{\mathsf{Label}\ G\ n = \texttt{callvirt}\ B\ c{::}\ell(A_1, \ldots, A_k) \quad n \neq \mathsf{Exit}\ G \quad c'[f_i \mapsto {u_i}^{i \in 1..n}] \in \mathsf{can\_read}\ t\ p \quad c' <: c}{t, G, h \vdash (s, vs; p; v_1; \ldots; v_k, args, n) \xrightarrow{\mathsf{read}\ t\ p\ c'[f_i \mapsto {u_i}^{i \in 1..n}]} (s; (vs, args, n), \cdot, .args(p, v_1, \ldots, v_k), \mathsf{next}\ G\ (\mathsf{mcall}\ c')\ n)}$$

Fig. 2. Some single-thread transition rules for GraphBIL

in the call stack. As in MiniLLVM, at each step we look up the instruction at the current node, perform some modification to the local state and/or produce some memory operations, and then move along an outgoing edge to a new node.

The concurrent semantics of GraphBIL are generated from the single-thread semantics as in MiniLLVM, with a concurrent state a pair of per-thread local state and a single shared memory:

$$\frac{t, \mathcal{G}_t, m \vdash states_t \xrightarrow{a} (s', vs', args', n') \qquad \mathsf{update\_mem}\ m\ a\ m'}{(states, m) \rightarrow (states(t \mapsto (s', vs', args', n')), m')}$$

4 Specifying Memory Models

In order to complete our language definitions, we must instantiate them with concurrent memory models. 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 [11], 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 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.

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

$$\begin{array}{l|l} \mathsf{write}\ \ell_1\ 1 & \mathsf{write}\ \ell_2\ 1 \\ x := \mathsf{read}\ \ell_2 & y := \mathsf{read}\ \ell_1 \end{array}$$

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

Fig. 3. Behavior forbidden by SC but allowed in TSO

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 3. 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 reads, allowing both reads to return 0. As shown by Sindhu et al. [20], 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 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.

5 Integrating External Analyses

Compiler optimizations are often conditioned on the results of a whole-program analysis. While many of these analyses can be expressed as CTL side conditions, we may also want to take advantage of the results of external analyses. One common example is *alias analysis*, which determines whether two pointer expressions may, must, or cannot refer to the same location in memory. Compilation frameworks such as LLVM [8] treat alias analysis as a black box and provide multiple alias analysis algorithms that can be selected at compile time, ranging from the minimally precise analysis that returns "may" for every query to more sophisticated algorithms. Various optimizations use the results of alias analysis, and their correctness does not depend on the details of the algorithm used (though their effectiveness might). Futhermore, static analyses are often highly dependent on the particular language under consideration; for instance, alias analysis on a register-machine language with variables is very different from alias analysis on a stack-machine language in which the only pointers are stored on the evaluation stack.

We can incorporate external analyses for a given target language through a two-step process. First, we axiomatize the analysis, describing the functions it must provide and the properties that those functions must satisfy. For instance, register-machine alias analysis provides one function – a function alias_analyze that takes a node in a tCFG and a pair of expressions and returns one of may, must, and cannot. The key property of this function is that if $\mathsf{alias\_analyze}(e, e')$ is must at p then e and e' must evaluate to the same location at p in any execution of the program, and if it is cannot then they must not. We might also impose further requirements: for instance, that two different global variables cannot alias.

Second, we must extend our set of atomic predicates with predicates that call the new functions. We can then refer to the results of the analysis in our CTL side conditions. Finally, if desired, we can implement the analysis and, by proving that the implementation satisfies the axioms, use it to obtain more concrete instances of our specifications. These implementations may be provided as algorithms, CTL side conditions, or general mathematical functions. In the formal semantics of PTRANS, we accomplish this axiomatization/implementation procedure using Isabelle's locale mechanism. For our case study, we axiomatize two external analyses for MiniLLVM – the alias analysis described above and a mutual exclusion (mutex) analysis – as well as an alias analysis for GraphBIL.

6 Verification

In this section, we will show the use of PTRANS in verifying optimizations. The candidate optimization is Redundant Store Elimination (RSE), which eliminates stores that are always overwritten before they are used. We will show the form that this optimization takes in both MiniLLVM and GraphBIL, and examine how its side conditions are affected by different memory models. Finally, we will outline the proofs of correctness for five different versions of the optimization, and remark on the amount of proof reuse enabled by our framework.

6.1 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, every graph G' output by applying T to G has some desired property relative to G. We will use *observational refinement* [4] 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* [3]:

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 relations we have presented for our target languages are 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 1. *Fix a memory model supporting the functions* $\mathsf{free\_set}$*,* $\mathsf{can\_read}$*, and* $\mathsf{update\_mem}$*. 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. $\mathsf{free\_set}\ m = \mathsf{free\_set}\ m'$
2. *For any* $u \neq t$*, if* $u, \mathcal{G}_u, m' \vdash s_1 \xrightarrow{a} s_2$*, then* $\mathsf{can\_read}\ m\ u\ \ell = \mathsf{can\_read}\ m'\ u\ \ell$ *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* $\mathsf{update\_mem}\ m'\ a\ m_2'$ *holds, then there exists some* m_2 *such that* $\mathsf{update\_mem}\ m\ a\ m_2$ *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.

6.2 Redundant Store Elimination for MiniLLVM

We can now attack the problem of proving the correctness of redundant store elimination (RSE) for MiniLLVM. The basic shape of the program fragment to be rewritten is shown in Figure 4. 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 in the original program no steps are possible, while eliminating s would allow the program to run until reaching s', potentially introducing new behavior.

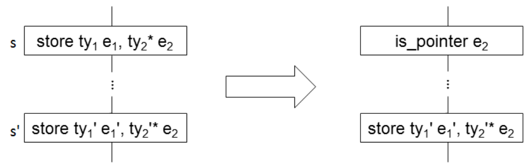

Fig. 4. Redundant Store Elimination

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 parallel case, the correctness condition is more complex, since changes to a memory location can be observed by other threads. While correctness can be ensured by requiring that no memory operations are performed between s and s' or by using mutual exclusion analysis to ensure that no other threads use the memory at e_2 before s', under relaxed memory models, the optimization can be more widely applied.

The "rewrite plus condition" style of PTRANS lends itself well to stating multiple versions of an optimization for different 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 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):

$$\mathsf{replace}\ n\ \mathsf{with}\ \mathtt{is\_pointer}\ e_2\ \mathsf{if}\ EF\ \mathsf{node}_t(n) \land \mathsf{stmt}_t(\mathtt{store}\ ty_1\ e_1, ty_2^*\ e_2) \land \varphi$$

Now, for each memory model, we need only provide the appropriate condition φ. 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. The optimization will be safe if removing the store at n does not expose any new behaviors, including those caused by other threads seeing shared values in a different order. Thus, 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. We will take the more complicated approach, using our 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 5, as well as a not_touches predicate that ensures that a given memory location is not read or modified by a given thread, the condition can be written as:

$$\begin{aligned}\varphi_{SC} \triangleq\ &\mathsf{protected}(x, e_2) \wedge \neg\mathsf{is}(x, e_2) \wedge A\ \mathsf{in\_critical}_t(x, e_2) \wedge (\mathsf{node}_t(n) \vee \mathsf{not\_touches}_t(e_2))\\ &\mathcal{U}\ (\mathsf{in\_critical}_t(x, e_2) \wedge \neg\mathsf{node}_t(n) \wedge \exists ty'_1\ e'_1\ ty'_2.\ \mathsf{stmt}_t(\texttt{store}\ ty'_1\ e'_1, ty'_2\ e_2))\end{aligned}$$

Next we will consider the appropriate side condition for the TSO memory model. Since TSO allows writes to be delayed past other operations, in a program with a redundant store, the redundant store may be delayed until immediately before the following store to e_2. If this behavior is possible in the original program, the observable behaviors of the transformed program will be a strict subset of those of the original program. Thus, we can use our side condition to describe 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. To construct the necessary side condition we must first define a predicate not_loads that is analogous to not_touches, but checks only for `load` instructions that might alias, rather than all memory operations. We also make use of a predicate not_mods that checks that the value of e_2 is not changed by the current instruction (i.e., the local variables that appear in e_2 are not modified). We can then construct the following condition:

$$\begin{aligned}\varphi_{TSO} \triangleq\ &AX_t(A\ \mathsf{not\_mods}_t(e_2) \wedge \mathsf{not\_loads}_t(e_2) \wedge \neg(\exists x\ ty_1\ e_1\ ty_2\ e'_2\ ty_3\ e_3.\ \mathsf{stmt}_t(\texttt{store}\ ty_1\ e_1, ty_2^*\ e'_2) \vee\\ &\quad \mathsf{stmt}_t(\%x = \texttt{cmpxchg}\ ty_1^*\ e_1, ty_2\ e'_2, ty_3\ e_3))\\ &\mathcal{U}\ (\neg\mathsf{node}_t(n)\ \wedge \exists ty'_1\ e'_1\ ty'_2.\ \mathsf{stmt}_t(\texttt{store}\ ty'_1\ e'_1, ty'_2\ e_2)))\end{aligned}$$

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 $\varphi_{SC} \vee \varphi_{TSO}$ 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:

$$\begin{aligned}\varphi_{PSO} \triangleq\ &AX_t(A\ \mathsf{not\_mods}_t(e_2) \wedge \mathsf{not\_touches}_t(e_2) \wedge \neg(\exists x\ ty_1\ e_1\ ty_2\ e'_2\ ty_3\ e_3.\\ &\quad \mathsf{stmt}_t(\%x = \texttt{cmpxchg}\ ty_1^*\ e_1, ty_2\ e'_2, ty_3\ e_3))\\ &\mathcal{U}\ (\neg\mathsf{node}_t(n)\ \wedge \exists ty'_1\ e'_1\ ty'_2.\ \mathsf{stmt}_t(\texttt{store}\ ty'_1 e'_1, ty'_2\ e_2)))\end{aligned}$$

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.

We are now ready to demonstrate the correctness of MiniLLVM RSE in PTRANS. As laid out in Section 6.1, 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 2. *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 1, 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 4. *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, and we can use this to explain the correctness of RSE under TSO and PSO:

Theorem 3. *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 1. 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 1 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 4. *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 1. 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. □

6.3 Redundant Store Elimination for GraphBIL

Redundant stores in a stack-machine language such as GraphBIL are somewhat less obvious than their counterparts in a register-machine language. In particular, since the operands of store instructions are implicitly drawn from the stack, there is much less indication in the program syntax of which stores might be redundant, as can be seen in Figure 5. However, by relying on the results of a stack-based alias analysis, we can perform a very similar optimization. Because mutual exclusion analysis is not obviously transferrable to the stack-machine context, we will limit ourselves to the TSO and PSO models; however, within these models, the specification of the optimization and the required simulation relations are remarkably close to their MiniLLVM counterparts.

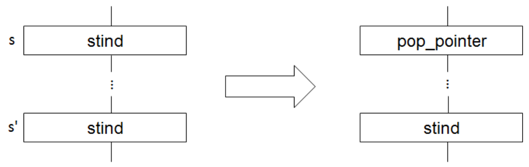

Fig. 5. Redundant `stind` Elimination

As before, we begin by stating an optimization skeleton that will be used across several memory models:

$$\text{replace } n \text{ with } \texttt{pop\_pointer} \text{ if } EF\ \text{node}_t(n) \wedge \text{stmt}_t(\texttt{stind}) \wedge \varphi$$

The GraphBIL counterpart to a MiniLLVM store is the `stind` instruction, which reads a location and a value from the evaluation stack and stores the value to the location. Just as in MiniLLVM, our condition under TSO must ensure that there are no reads from the location of the redundant store and no stores to any location between the first and the second store. Note that we must use alias analysis to ensure that the following `stind` instruction targets the same location as the original `stind`, since there is no variable whose value we can check is held constant from one to the other. Our stack-machine alias analysis predicates have the form $\text{must\_alias}_t(i, (n', i'))$ (and similarly for cannot_alias), where if $\text{must\_alias}_t(0, (n', 1))$ holds at a node n in t, then the element at the top of the stack at n must alias with the element 1 slot below the top of the stack at n' in all executions. We can use this to build a predicate $\mathsf{not\_loads}_t(n, i)$ that checks that no loads occur which could alias to stack slot i at node n, and define the condition as:

$$\begin{aligned}\varphi_{TSO} \triangleq\ & AX_t(A\ \mathsf{not\_loads}_t(n,1) \wedge \neg(\exists ty\ c\ m\ ty_1 ... ty_k.\ \mathsf{stmt}_t(\mathtt{stind}) \vee \\ & \mathsf{stmt}_t(\mathtt{callvirt}\ ty\ c{::}m(ty_1,\ ...,\ ty_k)) \vee \mathsf{stmt}_t(\mathtt{newobj\ void}\ c{::}\ \mathtt{.ctor}(ty_1,\ ...,\ ty_k)) \vee \mathsf{stmt}_t(\mathtt{box}\ c)) \\ & \mathcal{U}\ (\neg \mathsf{node}_t(n) \wedge \mathsf{stmt}_t(\mathtt{stind}) \wedge \mathsf{must\_alias}_t(1, (n, 1))))\end{aligned}$$

Under PSO, we can relax the condition on stores to other locations, and the condition is otherwise identical:

$$\begin{aligned}\varphi_{PSO} \triangleq\ & AX_t(A\ \mathsf{not\_loads}_t(n,1) \wedge \mathsf{not\_stores}_t(n,1) \wedge \neg(\exists ty\ c\ m\ ty_1 ... ty_k. \\ & \mathsf{stmt}_t(\mathtt{callvirt}\ ty\ c{::}m(ty_1,\ ...,\ ty_k)) \vee \mathsf{stmt}_t(\mathtt{newobj\ void}\ c{::}\ \mathtt{.ctor}(ty_1,\ ...,\ ty_k)) \vee \mathsf{stmt}_t(\mathtt{box}\ c)) \\ & \mathcal{U}\ (\neg \mathsf{node}_t(n) \wedge \mathsf{stmt}_t(\mathtt{stind}) \wedge \mathsf{must\_alias}_t(1, (n, 1))))\end{aligned}$$

We define φ'_{TSO} and φ'_{PSO} by stripping off the AX_t operators as above.

The structure of our proofs for GraphBIL is very close to that of the proofs for MiniLLVM. We present nearly identical candidate simulation relations, and divide the proof of single-thread simulation into four cases: case 1 of the simulation relation at the transformed node, case 1 of the simulation relation at other nodes, case 2 of the simulation at the following store, and case 2 of the simulation at a node between the transformed node and the following store. Each case follows by the same logic as in the MiniLLVM proofs.

Theorem 5. *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 location ℓ, 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.

Theorem 6. *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 location ℓ, $b_{t,\ell}$ 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.

6.4 Factoring Out Common Elements

One of the goals of VeriF-OPT is to simplify the process of verifying optimizations by making proofs as modular as possible. In particular, we aim to leverage the language-independence of the framework to state and prove general compiler verification principles, which can then be instantiated for any target language. The proof of correctness of each of the RSE optimizations relies on about 345 lemmas (excluding facts about infinite lists, general bisimulation, and other library functions/proofs). Of these, roughly:

- 55 were basic facts about the semantics of PTRANS.
- 90 were about graphs, CFGs, tCFGs, and paths through tCFGs.
- 30 were about step relations and simulations between graphs given step relations, including the theorems of Section 6.1 that let us lift single-thread simulations to concurrent programs.

These lemmas were reused for GraphBIL verification, and could be reused again in the verification of any optimization on any language under any memory model. Furthermore, roughly:

– 25 were about specific memory models, but were independent of the target language.
– 120 were about MiniLLVM, but were independent of the memory model.
– 10 were about MiniLLVM under particular memory models, but could be reused for any MiniLLVM optimization.
– 5 were specific to RSE in MiniLLVM, but were independent of the memory model.
– 10 were specific to the particular version of RSE under the particular memory model being verified.

Finally, the simulation relation and the structure of the proof of simulation for a given memory model in MiniLLVM needed only relatively small changes to be adapted for the same memory model in GraphBIL. In all, about half of the work was completely generalizable to other systems; of what remained, the majority was specific to the language being analyzed, but could be reused for any optimization in that language under any memory model, and proof techniques for one optimization in one language could be reused for a similar optimization in a markedly different language. This provides some evidence of the advantages of the language-independent approach, and we expect that as more target languages and optimizations are incorporated into the system, the library of reusable facts will continue to grow.

7 Conclusion and Future Work

In this paper, we present the PTRANS specification language for transformations on parallel programs, and show how it can be used to state and verify compiler optimizations in multiple languages and memory models. 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, and show that much of the proof effort can be reused for verifying a related optimization on a CIL-based stack-machine language, despite significant differences in syntax and semantics. We believe that the PTRANS methodology will simplify the process of stating and verifying optimizations on parallel code, by allowing for clean, proof-amenable statements of program transformations and emphasizing modular, reusable formalizations and proofs. Ultimately, we hope that the methodology here presented will aid compiler designers and formal methods researchers in creating new verified optimizations and compilers for a wide range of source, target, and intermediate languages.

The framework as described leaves plenty of room for future work. Since we have made it one of our primary goals to be able to state and verify optimizations involving concurrency, another clear path forward is to test our framework on a wider range of concurrency paradigms, such as the fork-join model used in C and Java. In order to state correct optimizations in PTRANS on programs with fork and join operations, we would need to write conditions that identified (perhaps approximately) which instructions could be guaranteed not to execute in more than one thread, and otherwise construct side conditions that, when checked on a single thread, ensured desirable runtime properties on all threads spawned by that thread – for instance, determining from the code of a single thread that all threads spawned will preserve mutual exclusion for a resource. There are no theoretical limitations barring us from expressing such properties in our framework, and trying to verify optimizations on, for instance, MiniLLVM with fork and join instructions would be an interesting case study and bring our target languages a step closer to the real world.

8 Related Work

Our work builds on the TRANS approach of expressing optimizations as rewrites on control flow graphs with temporal logic side conditions due to Lacey et al. [7] and Kalvala et al. [5]. The approach has been put into practice in several tools, including the Cobalt specification system [9], which also aims to prove the correctness of optimizations. While Cobalt provides fully automatic verification for its optimizations, this automation comes at the cost of expressiveness: Cobalt is limited to a much smaller set of CTL side conditions than TRANS in general. To the best of our knowledge, neither Cobalt nor any other work in

TRANS has sought to incorporate language-independence or awareness of concurrency. In previous work, we verified RSE on MiniLLVM and a simplified RSE on GraphBIL [14, 15], and gave an algorithm for executing PTRANS specifications as prototype optimizations [12].

There is considerably less work in the literature on verified stack-machine optimizations than on their register-machine counterparts. Notably, Saabas and Uustalu [18] have verified (with pen and paper) several optimizations for a simple stack-machine language, including dead store elimination and several other dead code elimination optimizations, by formalizing dataflow analyses in terms of type systems for both stack locations and local variables.

The CompCertTSO project [19], which aims to add support for concurrency to CompCert, involved the verification of several optimizations that could be lifted directly from single-threaded to concurrent programs, as well as a concurrency-specific fence elimination optimization [21]. The intermediate languages used are composed with an abstract machine implementing the TSO memory model, so that facts about TSO can be proved separately from facts about the intermediate languages. However, there is no evidence that the proofs of CompCertTSO can be generalized to other intermediate languages or other memory models. Ševčík [22] has also verified various optimizations, including redundant instruction eliminations, under data-race-free sequential consistency, specifying optimizations directly as transformations on the execution traces of programs. This approach is language-independent, but the optimizations modified may not correspond to any systematic modification of program code, and the focus is more on verifying transformations regardless of target language than on supporting the verification of language-specific optimizations.

Acknowledgements 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.

References

1. ECMA: ECMA-335: Common Language Infrastructure (CLI). ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA:adr (Dec 2001), `http://www.ecma.ch/ecma1/STAND/ecma-335.htm`
2. Gordon, A.D., Syme, D.: Typing a multi-language intermediate code. SIGPLAN Not. 36(3), 248–260 (Jan 2001), `http://doi.acm.org/10.1145/373243.360228`
3. Hennessy, M., Milner, R.: On observing nondeterminism and concurrency. In: de Bakker, J., van Leeuwen, J. (eds.) Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 85, pp. 299–309. Springer Berlin / Heidelberg (1980), `http://dx.doi.org/10.1007/3-540-10003-2\_79`, 10.1007/3-540-10003-2_79
4. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)
5. Kalvala, S., Warburton, R., Lacey, D.: Program transformations using temporal logic side conditions. ACM Trans. Program. Lang. Syst. 31(4), 1–48 (2009)
6. Krinke, J.: Context-sensitive slicing of concurrent programs. SIGSOFT Softw. Eng. Notes 28(5), 178–187 (Sep 2003), `http://doi.acm.org/10.1145/949952.940096`
7. Lacey, D., Jones, N.D., Van Wyk, E., Frederiksen, C.C.: Proving correctness of compiler optimizations by temporal logic. SIGPLAN Not. 37(1), 283–294 (Jan 2002), `http://doi.acm.org/10.1145/565816.503299`
8. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004), `http://dl.acm.org/citation.cfm?id=977395.977673`
9. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. SIGPLAN Not. 38, 220–231 (May 2003), `http://doi.acm.org/10.1145/780822.781156`
10. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. 43(4), 363–446 (Dec 2009), `http://dx.doi.org/10.1007/s10817-009-9155-4`
11. LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html (April 2014)
12. Mansky, W., Griffith, D., Gunter, E.L.: Specifying and executing optimizations for parallel programs Accepted for publication by GRAPHITE '14.

13. Mansky, W., Gunter, E.: A framework for formal verification of compiler optimizations. In: Proceedings of the First international conference on Interactive Theorem Proving. pp. 371–386. ITP'10, Springer-Verlag, Berlin, Heidelberg (2010), `http://dx.doi.org/10.1007/978-3-642-14052-5\_26`
14. Mansky, W., Gunter, E.L.: Verifying optimizations for concurrent programs, submitted to WPTE '14.
15. Mansky, W.E.: Specifying and Verifying Program Transformations with PTRANS. Ph.D. thesis, University of Illinois at Urbana-Champaign (May 2014)
16. Morisset, R., Pawan, P., Zappa Nardelli, F.: Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. SIGPLAN Not. 48(6), 187–196 (Jun 2013), `http://doi.acm.org/10.1145/2499370.2491967`
17. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems. pp. 151–166. Springer-Verlag, London, UK (1998)
18. Saabas, A., Uustalu, T.: Type systems for optimizing stack-based code. Electron. Notes Theor. Comput. Sci. 190(1), 103–119 (Jul 2007), `http://dx.doi.org/10.1016/j.entcs.2007.02.063`
19. Ŝevčik, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. SIGPLAN Not. 46(1), 43–54 (Jan 2011), `http://doi.acm.org/10.1145/1925844.1926393`
20. Sindhu, P.S., Frailong, J.M., Cekleov, M.: Formal specification of memory models. In: Dubois, M., Thakkar, S. (eds.) Scalable Shared Memory Multiprocessors, pp. 25–41. Springer US (1992), `http://dx.doi.org/10.1007/978-1-4615-3604-8_2`
21. Vafeiadis, V., Nardelli, F.Z.: Verifying fence elimination optimisations. In: Proceedings of the 18th international conference on Static analysis. pp. 146–162. SAS'11, Springer-Verlag, Berlin, Heidelberg (2011), `http://dl.acm.org/citation.cfm?id=2041552.2041566`
22. Ševčík, J.: Safe optimisations for shared-memory concurrent programs. SIGPLAN Not. 46(6), 306–316 (Jun 2011), `http://doi.acm.org/10.1145/1993316.1993534`
23. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. SIGPLAN Not. 46(6), 283–294 (Jun 2011), `http://doi.acm.org/10.1145/1993316.1993532`