Validating Optimizations of Concurrent C/C++ Programs

Soham Chakraborty Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS), Germany

{sohachak,viktor}@mpi-sws.org

Abstract

We present a validator for checking the correctness of LLVM compiler optimizations on C11 programs as far as concurrency is concerned. Our validator checks that optimizations do not change memory accesses in ways disallowed by the C11 and/or LLVM memory models. We use a custom C11 concurrent program generator to trigger multiple LLVM optimizations and evaluate the efficacy of our validator. Our experiments highlighted the difference between the C11 and LLVM memory models, and uncovered a number of previously unknown compilation errors in the LLVM optimizations involving the C11 concurrency primitives.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.4 [*Programming Languages*]: Processors; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Concurrency; Weak Memory Models; C/C++; LLVM; Translation Validation

1. Introduction

In our multicore era, exploiting concurrency is crucial in order to achieve good application performance. To facilitate this, programming languages have added first-class support for concurrency. For example, Java introduced volatile fields (Manson et al. 2005), while the 2011 C/C++ standards (henceforth, C11) introduced atomic variables and memory fences (ISO/IEC 9899:2011; ISO/IEC 14882:2011). These constructs provide a higher-level platform-independent abstraction over the concurrency semantics of existing multicore hardware implementations.

Nevertheless, for these constructs to *actually* facilitate concurrent programming, one must ensure that they are correctly compiled down to each hardware platform. This involves two tasks: (*i*) correctly mapping the language's concurrency primitives down to the corresponding hardware ones, and (*ii*) ensuring that standard compiler optimizations such as common subexpression elimination (CSE) do not falsify this mapping. The first task is mostly well understood: there are, for example, detailed mappings of the C11 atomic constructs to appropriate instruction sequences for x86 (Batty et al. 2011), and also for PowerPC and ARM (Sarkar et al. 2012; Batty et al. 2012), together with formal proofs that the mappings are correct.

```
              atomic_int lck = 0; int g = 0;
              lock() {...}
              unlock() {lck = 0; }

lock();    || lock();        ⇝   lock();    || lock();
g = 42;    || r = g;             unlock();  || r = g;
unlock();  || unlock();          g = 42;    || unlock();
```

Figure 1. Unsafe reordering introducing a data race on g.

The second task, however, is less clear. Compilers, such as GCC (`http://gcc.gnu.org/`) and LLVM (`http://llvm.org`), perform many optimizations that are correct for sequential programs, but whose correctness under concurrency has not formally been established.

As a very simple example of an incorrect transformation, consider the one shown in Figure 1. The program before the transformation always uses the shared variable g while the lock is held, and so the two accesses to g are ordered. After reordering the g = 42; and the `unlock`(); statements, however, the store to g is no longer protected by the lock and hence may race with the load on g in the second thread. While reordering g = 42; and `unlock`(); is correct for sequential programs, it is clearly wrong for concurrent programs because it introduces a data race and, as such, it is forbidden by the C/C++11 standards (ISO/IEC 9899:2011; ISO/IEC 14882:2011).

Because correctness of compiler optimizations under concurrency is still not very well understood, existing compilers are typically very conservative when encountering concurrency features and often miss optimization opportunities. Consider the following C++ code snippet, where

X is an atomic variable and two consecutive stores of order `std::memory_order_relaxed` (RLX) are performed.

$$\begin{array}{l}\texttt{X.store}(1, \mathsf{RLX});\\ \texttt{X.store}(3, \mathsf{RLX});\end{array} \rightsquigarrow \texttt{X.store}(3, \mathsf{RLX});$$

Although deleting the earlier store is correct (Vafeiadis et al. 2015), both GCC and LLVM do not currently do so.

Nevertheless, despite being conservative, compilers do have concurrency bugs. Morisset et al. (2013) reported a number of incorrectly introduced writes in GCC, while our current work has identified three previously unknown concurrency optimization errors in the LLVM *opt* phase:

(#22306) The SLP vectorizer violates reordering constraints.
(#22514) A combination of two *opt* transformations moves an access outside of a critical region similar to Figure 1.
(#22708) The "Global Value Numbering" (GVN) optimization performs an unsafe memory access reordering.

We expect that there will be many more such errors in compilers as they start becoming less conservative in optimizing concurrent code and perform optimizations such as redundant fence elimination (Vafeiadis and Zappa Nardelli 2011; Elhorst 2014). For this reason, we believe that verification will become key to avoiding errors in modern compilers.

Our goal in this work is to improve the compilation of concurrent programs by checking for concurrency compilation errors. We construct a validator[1] that checks whether the transformations performed by LLVM are correct according to the C11 memory model or our inferred LLVM memory model. The validator takes as inputs the programs before and after a set of transformations. It compares them by matching their memory access patterns and reports on whether it could find a matching demostrating that transformation is correct.

We have actually developed two matching algorithms: a compiler-independent and a compiler-specific one (see Section 3). These algorithms handle the access pattern changes caused by reorderings, eliminations, and introductions of memory accesses along with changes to the program's control flow graph (CFG) performed by LLVM. As a result, they are quite sophisticated. Our matching algorithms are sound for programs without loops; moreover, they return precise results when the values written on the shared locations are constants. Programs with loops are handled heuristically.

We have evaluated our validator using automatically generated C programs with many shared variable accesses, constructed so as to provide sufficient optimization opportunities to LLVM. Our tests highlight the relative merits and weaknesses of the two matching algorithms and have revealed a few bugs in the LLVM version 3.6, which we reported and were fixed in the next version 3.7rc2. Finally, we observe that there is an important semantic gap between the LLVM and C11 memory models, which affects the set of allowed optimizations and has been a source of compiler bugs.

[1] The validator is available at `http://plv.mpi-sws.org/validc/`.

2. The C11 and LLVM Memory Models

C11 and the LLVM intermediate representation (IR) have four kinds of instructions that affect memory: loads (R), stores (W), atomic updates (U), such as compare-and-swap and atomic increment, and memory fences (F). We call instructions of these kinds *actions*.

Following C11, the LLVM IR annotates each of these actions with a memory order, k. This can range from "non-atomic" (NA), the default for ordinary memory accesses, to "sequentially consistent" (SC), the default for atomic memory accesses. In between these two extremes, there are other possible memory orders: relaxed/monotonic (RLX), acquire (ACQ), release (REL), and combined release-acquire (REL-ACQ).[2] The memory orders are ordered in terms of strength: $k \sqsupseteq k'$ says that k is stronger than k'. This is defined as the smallest order containing $\text{SC} \sqsupseteq \text{REL-ACQ} \sqsupseteq \text{ACQ} \sqsupseteq \text{RLX} \sqsupseteq \text{NA}$ and $\text{REL-ACQ} \sqsupseteq \text{REL} \sqsupseteq \text{RLX}$.

We call an access *acquire* if its order is ACQ or stronger; namely, ACQ, REL-ACQ, or SC. We call it *release* if its order is REL or stronger; that is, REL, REL-ACQ, or SC.

2.1 C11 Semantics

We briefly discuss the semantics of C11 programs. Detailed formal expositions of the C11 memory model can be found in Batty et al. (2011) and Vafeiadis et al. (2015).

According to C11, program semantics is given by a set of consistent executions. Executions are directed graphs whose nodes are actions and whose edges describe the order of the actions in the program (the *program order*, *po*) and the write from which each read gets its value (the *reads from* relation). Whenever an acquire read "reads from" a release write, the release write *synchronizes with* the acquire read (sw). Happens-before is the least transitive relation that includes the program order and the synchronization relation (i.e., $hb = (po \cup sw)^+$). An execution is called *consistent* if it satisfies a bunch of conditions dictating, for example, that a read cannot read from a write that happens after it.

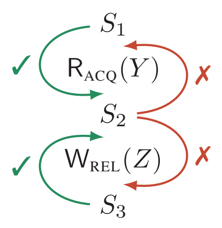
Two accesses are *concurrent* in an execution if they are not related by the happens-before order. An execution is *racy* if it has two concurrent accesses to the same location, at least one of which is non-atomic (NA) and at least one of which is a write or update. Programs that have a consistent racy execution have "undefined" semantics; otherwise their semantics is exactly the set of their consistent executions.

To illustrate the semantics consider the following example where all variables initially have the value zero.

$$\begin{array}{r@{}l} g_{\text{NA}} &= 3;\\ X_{\text{REL}} &= 1; \end{array} \;\Bigg\|\; \begin{array}{l} \mathsf{if}(X_{\text{ACQ}})\\ \quad g_{\text{NA}} = 4; \end{array}$$

This program is race-free because the only execution where both accesses to g occur is one where the X_{ACQ} reads the value 1. This can only happen if it reads from the X_{REL}

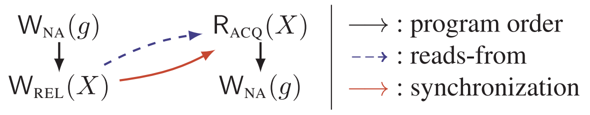
[2] C11 also supports "consume" memory accesses, but LLVM conflates them with acquire accesses.



The "roach motel" principle says that shared memory accesses can be moved in critical regions but not out of them (Manson et al. 2005). The analogue is that cockroaches (resp., accesses) can check in the motel (resp., the region), but not check out.

Figure 2. The "roach motel" reordering principle.

store in the first thread, which induces a synchronization between the two X accesses, and in turn a happens-before order between the two g accesses. This is depicted below:



$\mathsf{W}_{\text{NA}}(g)$
$\mathsf{R}_{\text{ACQ}}(X)$
$\mathsf{W}_{\text{REL}}(X)$
$\mathsf{W}_{\text{NA}}(g)$
$\longrightarrow$: program order
$\dashrightarrow$: reads-from
$\longrightarrow$: synchronization

The program would remain race free if we were to strengthen one or both of the X accesses to SC order; it would, however, become racy if we weaken one or both of the X accesses to RLX order. Besides the synchronizations between release writes and acquire reads, similar ones get induced by memory fences. For more details, see Batty et al. (2011).

2.2 Allowed Transformations in C11

We review the memory access reordering and elimination transformations allowed by C11. Memory access introduction is typically incorrect as it may introduce a data race. The only exception is introducing a read adjacent to another access of the same memory location.

Safe Reorderings In the sequential setting, two adjacent actions a and b are reorderable ($a; b \rightsquigarrow b; a$) if they access different locations and they do not have any control or data dependence between them. In the concurrent setting, we must further ensure that no accesses are moved out of critical regions as in Figure 1. The notions of *acquire* and *release* actions generalize the notions of acquiring a lock and releasing a lock (Vafeiadis et al. 2015) as explained in Figure 2.

Elimination of Redundant Accesses Redundant actions can be eliminated in C11 if they are made redundant by an adjacent memory access (e.g., a store overwritten by another one). If the eliminated access and the justifying access are not adjacent, then the elimination is valid if the access to be eliminated can be moved so as to become adjacent to the one justifying its elimination. Vafeiadis et al. (2015), for example, present these elimination rules for atomic accesses.

Read-after-Read: $\mathsf{R}_X(\ell); C; \mathsf{R}_Y(\ell) \rightsquigarrow \mathsf{R}_X(\ell); C$ is safe if $acquire \notin C$ and $X \sqsupseteq Y$.

Read-after-Write: $\mathsf{W}_X(\ell); C; \mathsf{R}_X(\ell) \rightsquigarrow \mathsf{W}_X(\ell); C$ is safe if $acquire \notin C$ and $X \sqsupseteq Y$.

Overwritten Write: $\mathsf{W}_Y(\ell); C; \mathsf{W}_X(\ell) \rightsquigarrow C; \mathsf{W}_X(\ell)$ is safe if $release \notin C$ and $X \sqsupseteq Y$.

For eliminating non-atomic accesses, Ševčík (2011) and Morisset et al. (2013) identified a weaker condition: C should contain no release-acquire pairs i.e., no sequences of a release instruction followed by an acquire instruction.

Read-Elimination: $a; C; \mathsf{R}_{\text{NA}}(\ell) \rightsquigarrow a; C$ is safe if a is $\mathsf{W}_{\text{NA}}(\ell)$ or $\mathsf{R}_{\text{NA}}(\ell)$ and C contains no release-acquire pairs.

Write-Elimination: $a; C; \mathsf{W}_{\text{NA}}(\ell) \rightsquigarrow C; \mathsf{W}_{\text{NA}}(\ell)$ is safe if a is $\mathsf{W}_{\text{NA}}(\ell)$ and C contains no release-acquire pairs.

Absence of a release-acquire pair between two non-atomic actions a and b ensures that in a race-free program, no conflicting write of another thread can happen after a and before b: it must either happen before or after both accesses.

2.3 Differences between the C11 and LLVM Models

We observe that there is a significant gap between the intended LLVM semantics and those of C11 when it comes to racy programs. This gap is best illustrated by LLVM bug #22514, which we show in Figure 3. For convenience, we use C syntax instead of LLVM IR syntax and annotate the relevant implicit memory orders on the accesses.

The source program is race-free when *flag* is false. This is because the only access of g in the second thread happens after the SC-read of X, that synchronizes with the SC-write of X in the first thread. Moreover, one can see that the only possible result for r_2 is the value 4.

Now consider a sequence of two transformations. The first transformation moves the access of g before the conditional. This introduces a race because now g is read before the synchronization, and is therefore incorrect according to the C11 model. LLVM argues that it is correct because even if the speculative load of g returns a garbage value due to the race, this does not matter because that value is not actually used by the computation (see LLVM documentation).

The second transformation is *repeated read elimination*, a simple special case of common subexpression elimination (CSE). Since g has been read before the X_{SC} access, it need not be read again. This transformation is valid under the C11 model, and is explicitly permitted by Ševčík (2011) and Morisset et al. (2013), but it is incompatible with the previous transformation. The resulting program after the two transformations is not only racy, but may return $r_2 = 0$ even under SC.

As a result of reporting this bug, the LLVM developers decided to restrict the second transformation rather than the first one, which means that the intended LLVM memory model is subtly different from the C11 model. In LLVM's model, read-write races are allowed, the non-atomic read returns an *undef* value. A write-write race, however, still results in undefined behavior.

As we have seen, this has implications on the set of allowed program transformations in the LLVM model. On the one hand, the compiler may introduce unused speculative loads as in transformation (1). On the other hand, it cannot

$$\begin{array}{r|l}
\multicolumn{2}{l}{\mathsf{int}\ g = 0;\ \ \mathsf{atomic\_int}\ X = 0;} \\
 & r_1 = 0; \\
 & \mathsf{if}(\mathit{flag}) \\
g_{\text{NA}} = 4; & \quad r_1 = g_{\text{NA}}; \\
X_{\text{SC}} = 1; & \mathsf{if}(X_{\text{SC}} == 1) \\
 & \quad r_2 = g_{\text{NA}}; \\
 & \mathsf{else}\ r_2 = 4;
\end{array}
\overset{(1)}{\rightsquigarrow}
\begin{array}{r|l}
\multicolumn{2}{l}{\mathsf{int}\ g = 0;\ \ \mathsf{atomic\_int}\ X = 0;} \\
 & r_1 = 0; \\
 & t_1 = g_{\text{NA}}; \quad \textit{// introduced} \\
g_{\text{NA}} = 4; & r_1 = (\mathit{flag})?\ t_1 : 0; \\
X_{\text{SC}} = 1; & t_2 = X_{\text{SC}}; \\
 & t_3 = g_{\text{NA}}; \\
 & r_2 = (t_2 == 1)?\ t_3 : 4;
\end{array}
\overset{(2)}{\rightsquigarrow}
\begin{array}{r|l}
\multicolumn{2}{l}{\mathsf{int}\ g = 0;\ \ \mathsf{atomic\_int}\ X = 0;} \\
 & r_1 = 0; \\
 & t_1 = g_{\text{NA}}; \\
g_{\text{NA}} = 4; & r_1 = (\mathit{flag})?\ t_1 : 0; \\
X_{\text{SC}} = 1; & t_2 = X_{\text{SC}}; \\
 & \textit{// deleted } t_3 = g_{\text{NA}}; \\
 & r_2 = (t_2 == 1)?\ t_1 : 4;
\end{array}$$

Figure 3. A sequence of LLVM transformations: (1) introduce a speculative read of g during CFG simplification, (2) remove redundant read of g by the GVN pass. The composition violates the "roach motel" property when $\mathit{flag} = \mathit{false}$.

$$\begin{array}{l}
\textbf{for}\ (i = 0\,;\ i < 4\,; \\
\quad i{+}{+})\{ \\
\quad g[i]_{\text{NA}} = 0; \\
\quad X[i]_{\text{SC}} = i; \\
\}
\end{array}
\rightsquigarrow
\begin{array}{l}
X[0]_{\text{SC}} = 0; \\
X[1]_{\text{SC}} = 1; \\
X[2]_{\text{SC}} = 2; \\
g[0:3]_{\text{NA}} = 0; \\
X[3]_{\text{SC}} = 3;
\end{array}
\quad
\begin{array}{l}
\textbf{Context:} \\
\left[\begin{array}{c|l} - & \textbf{if}\ (X[2]_{\text{SC}}) \\ & \quad r = g[2]_{\text{NA}}; \end{array}\right]
\end{array}$$

Figure 4. SLP vectorizer performs an unsafe reordering.

$$\begin{array}{l}
\textbf{if}\ (\mathit{flag})\ \{ \\
\quad g_{\text{NA}} = 5; \\
\} \\
r = X_{\text{ACQ}}; \\
r' = (r\ ?\ g_{\text{NA}} : 8);
\end{array}
\rightsquigarrow
\begin{array}{l}
\textbf{if}\ (\mathit{flag})\ \{ \\
\quad g_{\text{NA}} = 5; \\
\quad t = 5; \\
\}\ \textbf{else}\ \{\ t = g_{\text{NA}};\ \} \\
r = X_{\text{ACQ}}; \\
r' = (r\ ?\ t : 8);
\end{array}
\quad
\begin{array}{l}
\textbf{Context:} \\
\left[\begin{array}{c|l} - & g_{\text{NA}} = 8; \\ & X_{\text{REL}} = 1; \end{array}\right]
\end{array}$$

Figure 5. GVN performs an unsafe reordering.

eliminate the 'redundant' non-atomic loads as can be eliminated in C11 by appealing to the data race freedom (DRF) property. Hence the read elimination rule becomes:

Read-Elimination: $a; C; \mathsf{R}_{\text{NA}}(\ell) \rightsquigarrow a; C$ where a is $\mathsf{W}_{\text{NA}}(\ell)$ or $\mathsf{R}_{\text{NA}}(\ell)$ and $\mathit{acquire} \notin C$.

2.4 Other Concurrency Compilation Errors

We now discuss the other concurrency-related LLVM optimization errors we discovered using our validator.

Bug #22306 The "Superword-Level Parallelism" (SLP) vectorizer performs the unsafe transformation shown in Figure 4 unrolling the loop and combining the four g accesses. Consider running the code in the context shown in the figure with all variables initialized to 0. The source program is race-free because the $X[2]_{\text{SC}}$ accesses synchronize and therefore the $g[2]_{\text{NA}} = 0$ *happens before* the load of $g[2]_{\text{NA}}$ in the second thread. The target program, however, contains a race between the $g[2]_{\text{NA}}$ load and the $g[0:3]_{\text{NA}}$ store.

Bug #22708 The "Global Value Numbering" (GVN) pass performs the transformation shown in Figure 5. Assume that the code is executed in the concurrent context shown in the figure and that all variables are initialized to 0: in particular, $\mathit{flag} = \mathsf{false}$.

The source program is race-free and can return only $r' = 8$ because if the program reads $X_{\text{ACQ}} \neq 0$, then it synchronizes with the parallel thread and sees the $g_{\text{NA}} = 8$ store. The target program, however, is racy and can return $r' = 0$ even with an interleaving semantics.

Summary We observe that all the bugs found violate the 'roach motel' principle. Bugs #22514 and #22708 reorder a load before an acquire command, whereas bug #22306 reorders a memory access after a release command. We also note that bugs #22514 and #22708 also introduced an unused load on certain paths, which is disallowed by C11 but allowed by LLVM.

3. Our Validation Approach

In this section, we describe our approach for validating LLVM optimizations with respect to concurrency.

The validation reads the source and the target programs and the memory model under which to perform the validation: C11 or LLVM. The validator then compares the programs by matching the shared memory accesses to identify how the transformation has affected the shared memory access sequences, and returns one of three results:

Correct: A safe matching was found between the source and the target witnessing the correctness of the transformation. This means that if we execute the target program and record the sequence τ of its memory accesses, then either the source program has undefined semantics or there is a way of executing it and obtaining a corresponding sequence σ of memory accesses that can be transformed into τ by performing the safe introduction, reordering and elimination rules of Sections 2.2 and 2.3.

Possible Error: There exists no safe match between the source and the target. We also report the cause(s) of error:

- Deletion of non-deletable accesses from the source;
- Incorrect reordering;
- Introduction of an observable write or update; or
- In case of C11, introduction of a potentially racy read operation.

Unknown: If the source program has any loop and the loop condition changes by any transformation (e.g. in loop unrolling) then the validator returns "unknown" since it does not handle such transformations.

We propose two approaches for performing the matching:

- *Compiler-independent matching (CIM).* In this scheme, the validator has no knowledge about how the memory accesses have been moved by the optimization. Thus, given the source and target access sequences, it tries to match them as precisely as possible.
- *LLVM-specific matching with instruction metadata (MD).* In this approach, we instrument the compiler so as to witness the movement of the shared memory accesses, thereby greatly simplifying the matching.

In both cases, we first map LLVM instructions to the actions defined in Section 2. In this mapping, we produce actions only for potentially shared accesses (i.e., accesses to global variables), not for accesses to registers or temporaries. We will now discuss the two approaches in detail.

3.1 Compiler Independent Matching (CIM)

In this approach, given the source and the target memory accesses we attempt to come up with a matching to check if the target is generated by a sequence of correct transformations.

We first explain how to detect if an action is redundant (§3.1.1). Then we discuss how to match the accesses on straight-line code (§3.1.2). Later we will discuss how to match programs with control flow (§3.1.3).

3.1.1 Marking Actions

Given the source action sequence, we categorize the actions as *non-deletable*(✓), *conditionally deletable*(⊗) or *immediately deletable*(✗). Non-deletable actions are those that cannot be deleted after any set of safe reordering or deletion transformations. Conditionally deletable actions may be removed only after some other safe transformation is applied. We explain the markings on the following sequence.

$$\begin{array}{rcl} \textbf{source} & & \textbf{target} \\ \otimes_{\mathsf{C}_1}\ a : \mathsf{W}_{\text{RLX}}(X) & & a' : \mathsf{W}_{\text{RLX}}(X) \\ \checkmark\ b : \mathsf{W}_{\text{REL}}(Y) & \rightsquigarrow & c' : \mathsf{W}_{\text{RLX}}(X) \\ \text{✗}\ c : \mathsf{W}_{\text{RLX}}(X) & & b' : \mathsf{W}_{\text{REL}}(Y) \\ \checkmark\ d : \mathsf{W}_{\text{RLX}}(X) & & d' : \mathsf{W}_{\text{RLX}}(X) \end{array}$$

$$\mathsf{C}_1 = [a; \mathsf{W}_{\text{RLX}}(X)]$$

Actions b and d are non-deletable because they are the last writes to X and Y in the source sequence. Action c is directly deletable because it immediately precedes another similar store to X. In contrast, a is only conditionally deletable: in order to be deleted, a later relaxed store to X must be reordered before the release store to Y to satisfy the condition C_1; e.g. $b; c \rightsquigarrow c'; b'$.

Similar notions of deletable and non-deletable actions also appear in earlier work (Ševčík 2011; Morisset et al. 2013; Vafeiadis et al. 2015) but require adaptation to our setting of checking for the validity of an unknown sequence of transformations.

Insufficiency of Release-Acquire Pairs Section 2.2 introduced the lack of release-acquire pairs as a way of identifying deletable operations. We observe that presence of a release-acquire pair does not entail that an access is non-deletable. Consider the following example.

$$\begin{array}{lclcl} \textbf{source} & & & & \textbf{target} \\ \otimes_{\mathsf{C}}\ a : \mathsf{W}_{\text{NA}}(g) & & \otimes_{\mathsf{C}}\ a : \mathsf{W}_{\text{NA}}(g) & & \checkmark\ c : \mathsf{R}_{\text{ACQ}}(Y) \\ \checkmark\ b : \mathsf{W}_{\text{REL}}(X) & \rightsquigarrow & \checkmark\ c : \mathsf{R}_{\text{ACQ}}(Y) & \rightsquigarrow & \text{✗}\ a : \mathsf{W}_{\text{NA}}(g) \\ \checkmark\ c : \mathsf{R}_{\text{ACQ}}(Y) & & \checkmark\ b : \mathsf{W}_{\text{REL}}(X) & & \checkmark\ d : \mathsf{W}_{\text{NA}}(g) \\ \checkmark d : \mathsf{W}_{\text{NA}}(g) & & \checkmark\ d : \mathsf{W}_{\text{NA}}(g) & & \checkmark\ b : \mathsf{W}_{\text{REL}}(X) \end{array}$$

$$\mathsf{C} = [a; \mathsf{W}_{\text{NA}}(g)]$$

In the source program, action a cannot be directly eliminated because there is a release-acquire pair between a and d. If, however, we transform the program by moving the acquire load earlier and/or the release store later, then in the target program, a may be removed. For this reason, we have to mark a as conditionally deletable in the source program.

C11 Release Sequences There is another subtlety in detecting deletable actions. Consider the program:

$$X_{\text{REL}} = 1; X_{\text{RLX}} = 2; X_{\text{REL}} = 3;$$

The first access to X is not deletable because according to C11, if an acquire read reads from the $X_{\text{RLX}} = 2$ store, then it synchronizes with the earlier $X_{\text{REL}} = 1$ store. Removing the $X_{\text{REL}} = 1$ store therefore removes a possible synchronization and is unsound. It is, however, conditionally deletable because if the $X_{\text{RLX}} = 2$ is deleted or strengthened to REL order, then the $X_{\text{REL}} = 1$ store can also be removed.

Synchronization Access Deletion We call *CAS*, *release*, *acquire* accesses synchronization actions. Even though according to the rules in §2.2 redundant synchronization actions may be eliminated under certain conditions, removing them goes against the programmer's intentions to communicate and synchronize across threads. Also removing such synchronization accesses can cause a deadlock or significantly degrade performance (e.g., by converting a test-and-test-and-set lock into a test-and-set lock). Moreover, currently neither LLVM nor GCC removes any atomic accesses. We therefore consider all such actions to be non-deletable.

Marking Algorithm Initially, we mark all actions to be *non-deletable* and proceed to mark individual actions as deletable(✗) or conditionally deletable(⊗). For example,

- In the sequence $a : \mathsf{W}_{\text{NA}}(\ell);\ C;\ b : \mathsf{W}_{\text{NA}}(\ell)$, a is ⊗ if C contains a release-acquire pair and deletable otherwise.
- In the sequence $a : \mathsf{R}_{\text{NA}}(\ell);\ C;\ b : \mathsf{R}_{\text{NA}}(\ell)$, In C11 b is ⊗ if C contains a release-acquire pair and ✓ otherwise. In LLVM b is ⊗ if *acquire* $\in C$ and ✓ otherwise.

3.1.2 Matching Access Sequences

We extract the source and target actions (indexed from 1 to N) as described before, mark them as discussed in Section 3.1.1, match them in multiple iterations, and finally analyze whether the matching denotes a correct transformation. We match the source and target actions in three steps:

1. **Synchronization actions.** We traverse the source and the target sequences from index 1 to N and match the synchronization actions. If any synchronization action remains unmatched or the matching is unsafe, we report "Possible Error" and return.
2. **Other non-deletable (✓) actions.** For each unmatched non-deletable source action we identify the matching window, i.e. the target subsequence within which a safe matching can occur. A matching outside the window implies that the access is unsafely reordered.

   For each non-deletable source action s, let a and b be the nearest predecessor and successor source actions such that $a; s \not\rightsquigarrow s; a$ (i.e., the $a; s \rightsquigarrow s; a$ is unsafe) and $s; b \not\rightsquigarrow b; s$ and a and b are matched with the j^{th} and k^{th} target actions respectively. The search window for s is then the target subsequence from $j + 1$ to $k - 1$. If s is a write or a release fence action then we search from the end to the start of the window and if s is a read or acquire fence action then we search from the start to the end of the window. If s remains unmatched, report "Possible Error" and return.
3. **Remaining unmatched target actions.** Since a transformation need not delete all of the deletable(✗) and conditionally deletable(⊗) source actions, there may still be unmatched actions in the target sequence. To match those unmatched target actions, again we identify the appropriate search windows in the source sequence within which a safe matching may be found.

   For each unmatched target action t, let a' and b' be the nearest predecessor and successor target actions such that $t; a' \not\rightsquigarrow a'; t$ and $b'; t \not\rightsquigarrow t; b'$ and a' and b' are matched to the j^{th} and k^{th} source actions respectively. The window for t is the source subsequence from $j+1$ to $k-1$. As before, for writes and release fences, we search the window from end to start, whereas for reads and acquire fences, from start to end. If t remains unmatched, we consider the access as *introduced* and analyze if the introduction of t is a safe transformation.

   Note that writes and release fences are matched from end to start in both the target and the source sequences, whereas reads and acquire fences are matched from start to end. Because of this, some earlier writes and later reads may remain unmatched, but in the subsequent analysis these may be considered as redundantly introduced actions and we report no error. If we attempted to match them in the reverse order, we would find matches for the redundant target accesses but fail to match the non-redundant ones, leading to an incorrect matching.

Once the analysis is complete, we analyze the unmatched actions as follows.

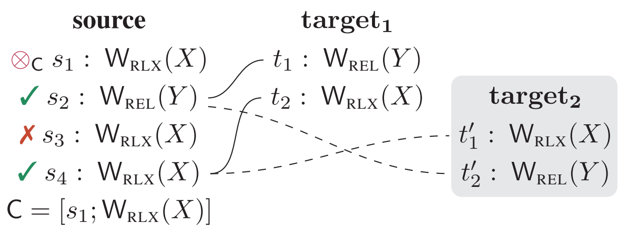
Analyze Introduced Actions An unmatched action in the target is an introduced action. We have three cases:

Writes/Updates: Introducing atomic writes or updates is generally unsound because a parallel thread may observe the additional update. However, introducing an immediately deletable non-atomic write is safe because any program that could observe the difference is anyway racy.

Reads: As bug #22514 shows, an introduced read is incorrect in C11, but allowed in our inferred LLVM model.

Fences: This is safe as it just adds synchronization.

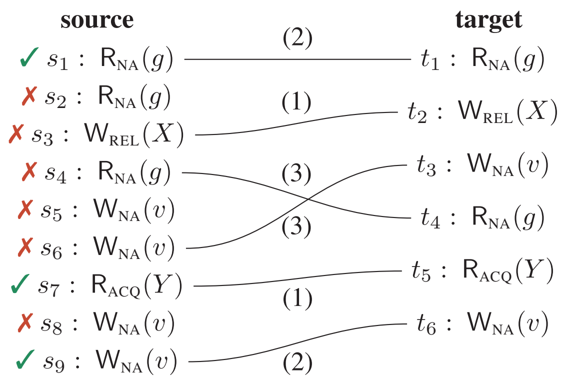
Analyze Deleted Actions Unmatched actions in the source signify actions that have been deleted. For immediately deletable actions, there is nothing to check. For conditionally deletable ones, we check that the deletion preconditions are met. As the following example shows, checking the deletion preconditions is sometimes a bit subtle.



source
$\otimes_{\mathsf{C}}\ s_1 : \mathsf{W}_{\text{RLX}}(X)$
✓ $s_2 : \mathsf{W}_{\text{REL}}(Y)$
✗ $s_3 : \mathsf{W}_{\text{RLX}}(X)$
✓ $s_4 : \mathsf{W}_{\text{RLX}}(X)$
$\mathsf{C} = [s_1; \mathsf{W}_{\text{RLX}}(X)]$
target$_1$
$t_1 : \mathsf{W}_{\text{REL}}(Y)$
$t_2 : \mathsf{W}_{\text{RLX}}(X)$
target$_2$
$t'_1 : \mathsf{W}_{\text{RLX}}(X)$
$t'_2 : \mathsf{W}_{\text{REL}}(Y)$

In this example, s_1 is conditionally deletable if C is satisfied. The action s_1 can be deleted only if s_2 and s_3 were reordered, but instead s_3 has been deleted! We therefore have to consider whether s_3 could have been reordered with s_2 before being deleted. In $\mathbf{target_1}$ the answer is *no* because after the reordering, s_3 is no longer deletable. Given that s_3 is deleted, we instead have to check that s_2 has been reordered with s_3's justifier (namely, s_4). So, the elimination of s_1 is *correct* in $\mathbf{target_2}$ but not in $\mathbf{target_1}$.

Example Finally we demonstrate the matching and analysis procedures on the following example:



source
✓ $s_1 : \mathsf{R}_{\text{NA}}(g)$
✗ $s_2 : \mathsf{R}_{\text{NA}}(g)$
✗ $s_3 : \mathsf{W}_{\text{REL}}(X)$
✗ $s_4 : \mathsf{R}_{\text{NA}}(g)$
✗ $s_5 : \mathsf{W}_{\text{NA}}(v)$
✗ $s_6 : \mathsf{W}_{\text{NA}}(v)$
✓ $s_7 : \mathsf{R}_{\text{ACQ}}(Y)$
✗ $s_8 : \mathsf{W}_{\text{NA}}(v)$
✓ $s_9 : \mathsf{W}_{\text{NA}}(v)$
(2)
(1)
(3)
(3)
(1)
(2)
target
$t_1 : \mathsf{R}_{\text{NA}}(g)$
$t_2 : \mathsf{W}_{\text{REL}}(X)$
$t_3 : \mathsf{W}_{\text{NA}}(v)$
$t_4 : \mathsf{R}_{\text{NA}}(g)$
$t_5 : \mathsf{R}_{\text{ACQ}}(Y)$
$t_6 : \mathsf{W}_{\text{NA}}(v)$

First, we mark the source and the target actions as discussed in Section 3.1.1 and then we match the synchronization source actions to the respective target actions. Thus we match s_3 with t_2 and s_7 with t_5. We proceed to the second step with the remaining non-deletable source actions. For s_1, the window is the singleton set $\{t_1\}$; so we match it with t_1.

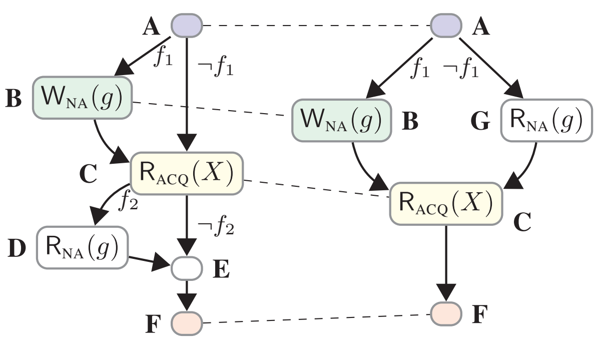

A
f_1
$\neg f_1$
B
$\mathsf{W}_{\mathrm{NA}}(g)$
C
$\mathsf{R}_{\mathrm{ACQ}}(X)$
f_2
$\neg f_2$
D
$\mathsf{R}_{\mathrm{NA}}(g)$
E
F
A
f_1
$\neg f_1$
$\mathsf{W}_{\mathrm{NA}}(g)$
B
G
$\mathsf{R}_{\mathrm{NA}}(g)$
$\mathsf{R}_{\mathrm{ACQ}}(X)$
C
F

Figure 6. Transformation of the program in Figure 5.

For s_9, the window likewise is the singleton set $\{t_6\}$; so we match it with t_6.

Finally, we try to match the remaining unmatched target actions, t_3 and t_4. To compute the search window, we identify the immediate predecessor *release* and immediate successor *acquire* of t_3 and t_4 which are t_2 and t_5 respectively. Thus, we match t_3 with s_6 and t_4 with s_4.

After the matching, we analyze the unmatched actions s_2, s_5, s_8. Since all three actions are *immediately deletable*, we conclude that deleting them is valid and hence the transformation is correct.

3.1.3 Dealing with Control Flow

We have so far discussed access matching for straightline code. In case of programs with control flow, there is more work to do. We proceed in two steps.

First, for each (loop-free) path in the target we identify the corresponding set of paths in the source. As we will explain, this is nontrivial because transformations may restructure the control flow, making it difficult to identify the corresponding source path for a given target path.

Second, for each pair of source and target paths, we identify the sequence of shared memory accesses and apply the matching discussed in Section 3.1.2.

The validation is "Correct" if every target path is correctly matched by *all* corresponding source paths. Otherwise, the validator reports "Possible Error." We first provide a simple control flow graph (CFG) matching example and then discuss the general approach.

Example Figure 6 presents the CFGs corresponding to transformation of Figure 5 that witnesses the GVN bug. To make the two CFGs have matching entry and exit blocks, we append to both CFGs the empty **F** block. We represent the branch conditions in **A** and **C** by f_1 and f_2 respectively.

The blocks **D** and **E** are deleted and block **G** is introduced. The **A**, **B C**, and **F** basic blocks are matched because their name and path conditions match.

The target paths and the corresponding source paths are:

$$\mathbf{ABCF} \rightarrow \{\mathbf{ABCDEF}, \mathbf{ABCEF}\} \text{ and}$$
$$\mathbf{AGCF} \rightarrow \{\mathbf{ACDEF}, \mathbf{ACEF}\}\,.$$

Among the corresponding path pairs, matching the accesses of **AGCF** and **ACDEF** yields an error because of the unsafe reordering $\mathsf{R}_{\mathrm{ACQ}}(X); \mathsf{R}_{\mathrm{NA}}(g) \not\rightsquigarrow \mathsf{R}_{\mathrm{NA}}(g); \mathsf{R}_{\mathrm{ACQ}}(X)$.

Now we discuss the control flow matching technique.

Restructured CFG Matching Let $\{B_1 \ldots B_n\}$ be the set of basic blocks where B_1 and B_n are the entry and exit blocks respectively in the CFG. Also $\{f_1 \ldots f_{n-1}\}$ be the set of respective branch conditions of blocks B_1 to B_{n-1}. Given a path $P = B_1; \ldots B_j; B$ the path condition of P is denoted by $[\![P]\!] = f_1 \wedge \ldots \wedge f_j$.

Now consider that $\{P_1, \ldots, P_k\}$ be the set of paths from B_1 to B. We say that the reachability condition of B is $\Upsilon(B) = [\![P_1]\!] \vee \ldots \vee [\![P_k]\!]$. For example, $\Upsilon(\mathbf{E}) = (f_1 \wedge f2) \vee (f_1 \wedge \neg f_2) \vee (\neg f_1 \wedge f2) \vee (\neg f_1 \wedge \neg f_2)$ in Figure 6. A basic block B is not reachable if $\Upsilon(B) = false$.

We observe that even if an optimization restructures the CFG, the basic block names and the reachability conditions across the transformation is preserved by LLVM. Based on this observation the matching algorithm works as follows.

1. Match the basic blocks of the source and target CFGs by name and the respective reachability conditions.
2. Enumerate the set of paths from the function's entry node to the exit node in the target CFG.
3. For each such path, we find the set of corresponding source paths. Formally, we say that two paths P_S and P_T are corresponding if and only if their projections to the matched basic blocks are equal, $P_S|_{\mathsf{matched}} = P_T|_{\mathsf{matched}}$ as well as $[\![P_T]\!] \rightarrow [\![P_S]\!]$.

Once a path pair is found, we proceed to the access sequence matching algorithm of Section 3.1.2.

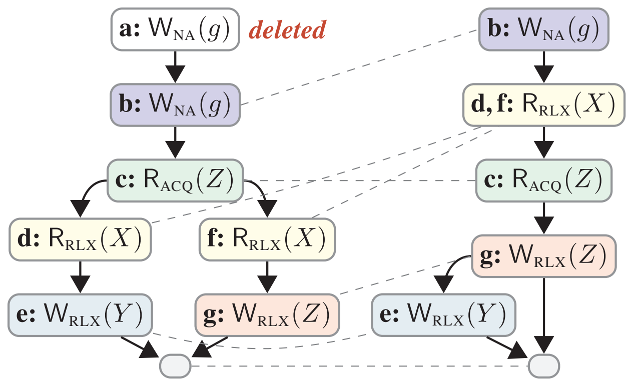
Path matching in the presence of loops is well known to be difficult (Sharma et al. 2013). We handle a loop heuristically. We unroll the loop body once and then cut-off the loop backedge. Effectively this is similar to considering that the loop has at most two iterations. This suffices to preserve the cross iteration reachability among the scalar accesses.

3.2 LLVM-specific Matching Using Metadata (MD)

Our second approach uses a specific feature of the LLVM IR. The LLVM IR allows one to attach arbitrary "metadata" information along with the program constructs to preserve debugging information throughout compilation without affecting the optimization. We use instruction metadata to keep track of how the actions are moved by a transformation.

In brief, we instrument each action in the source program with a uniquely named metadata node. Next, we run the optimization pass(es) on the instrumented source program. The attached metadata nodes do not affect the optimizations but are preserved by LLVM's code motion transformations. Afterwards we use the metadata nodes to match the memory accesses and check that the transformation is correct.

In more detail, we have mildly modified LLVM to attach a unique `MDNode` metadata node at each shared memory



a: $\mathsf{W}_{\text{NA}}(g)$
deleted
b: $\mathsf{W}_{\text{NA}}(g)$
c: $\mathsf{R}_{\text{ACQ}}(Z)$
d: $\mathsf{R}_{\text{RLX}}(X)$
f: $\mathsf{R}_{\text{RLX}}(X)$
e: $\mathsf{W}_{\text{RLX}}(Y)$
g: $\mathsf{W}_{\text{RLX}}(Z)$
b: $\mathsf{W}_{\text{NA}}(g)$
d, f: $\mathsf{R}_{\text{RLX}}(X)$
c: $\mathsf{R}_{\text{ACQ}}(Z)$
g: $\mathsf{W}_{\text{RLX}}(Z)$
e: $\mathsf{W}_{\text{RLX}}(Y)$

Figure 7. An unsafe transformation with matched accesses.

access and fence at the beginning of the *opt* phase before any transformation takes place. Optimization passes are, in principle, allowed to drop any metadata nodes attached to instructions and even to arbitrarily change them. In practice, however, all the *opt* transformations neither depend on nor alter any metadata that they do not recognize. As a result, the transformations tend to move instructions along with their attached metadata. A couple of transformations drop unrecognized metadata and we have modified them so as to preserve it. The modified LLVM also merges the metadata when multiple identical instructions are combined (e.g., in Simplify CFG). Finally, when a new instruction replaces an old instruction (e.g., in GVN), the metadata node is recreated from the old instruction.

3.2.1 Analysis of Matching

To illustrate the identification of errors after a metadata-based matching has been found, consider the dubious transformation in Figure 7. This transformation is incorrect for two reasons: (*i*) it violates the "roach motel" principle by reordering **c** and **d/f**; and (*ii*) it introduces a $\mathsf{W}_{\text{RLX}}(Z)$ on a path where it previously did not appear. We will now explain how to catch these two errors in turn.

Reordering Correctness To catch the incorrect reordering of **c** and **d/f** in Figure 7, for any two non-commuting matched accesses a and b (i.e., when $a; b \not\rightsquigarrow b; a$), we have to check that if a precedes b in the source CFG, then a still precedes b in the target CFG. So, for a given CFG G, we define the set $\mathsf{OrderedPairs}(G)$ of all (a, b) such that a and b are both matched actions, there is a path from a to b in G, and $a; b \not\rightsquigarrow b; a$. We then check that $\mathsf{OrderedPairs}(\mathsf{CFG}_{\mathsf{src}}) \subseteq \mathsf{OrderedPairs}(\mathsf{CFG}_{\mathsf{tgt}})$.

Returning to the graphs in Figure 7, this check fails for nodes **c** and **d/f** indicating that the reordering is unsafe.

Matched Access Movement Correctness The second error in the transformation in Figure 7 cannot be caught with the previous analysis. The movement of the **g** access is incorrect not because it violates any reordering constraints but rather because it introduces a write along the **c** $\rightarrow$ **e** path. To catch these errors we compare the path conditions of the source and the target. If the source and target path conditions are different and the access is observable in another thread, then we report "Possible Error" and "Correct" otherwise.

Returning to our example in Figure 7, we deduce that the movement of **g** is incorrect because the entry block is not similar to the one on the left branch of the conditional. The merging of the **d** and **f** accesses is, however, correct in this sense because $\Upsilon(P_d) \vee \Upsilon(P_f) = \Upsilon(P_{d,f})$.

Introduced Actions If the target program has any observable unmatched write or update actions or, in the case of C11, also any unmatched reads, we report "Possible Error" considering such accesses as (incorrectly) introduced.

Deleted Actions If the source program has any unmatched action, we have to check that their deletion is justified.

If the path condition is *false* then the action is not reachable and the deletion is justified. Otherwise, if the action is eliminable along every path from the entry node to the action then the deletion is correct and otherwise "Possible Error".

4. Evaluation and Discussion

We have implemented the two matching algorithms described in Section 3 and have applied them to validate transformations performed by LLVM. The results of our experiments are reported in Table 1.

4.1 Experimental Setup

Test Case Generation To evaluate our validator, we developed a randomized test case generator that constructs programs with a desired number of accesses, approximate proportions of each access type, and so on. Each generated program consists of a single function containing multiple accesses of global atomic and non-atomic scalar variables intertwined with some local computations and random control flow determined by boolean variables. Initially, we synthesized four classes of tests:

(a) straightline programs,
(b) programs with dead-path-free conditional control flow,
(c) programs with conditionals including dead paths, and
(d) programs containing conditionals and *do-while* loops.

For each class, we generated 100 programs with 100 shared memory accesses each (roughly 85% non-atomic and 15% atomic) and, where applicable, 10 branch conditions. We restricted the shared memory accesses to only five global variables so that the compiler has plenty optimization opportunities. In fact, in all the generated programs, LLVM successfully performed some optimization to them. To ensure that there are no dead paths in case (b), we generate a different flag as the guard for each conditional. In cases (c) and (d), with high probability we reuse the same flag for multiple conditionals so that the compiler may recognize and eliminate some dead paths.

For test cases (a)–(d), we used a large number of memory accesses to test the efficacy of our validator and ensure that

Test Class	Model	End-to-End Validation				Stepwise Validation					
		llvm 3.6		llvm 3.7rc2		llvm 3.6			llvm 3.7rc2		
		CIM	MD	CIM	MD	Non-id	CIM	MD	Non-id	CIM	MD
(a) Straightline	LLVM	95	95	0	0	927	95 †	95 †	835	0	0
	C11	0	0	0	0		0	0		0	0
(b) With Branches	LLVM	64	74	0	3	1209	64 †	74 †‡	1202	0	0
	C11	13	39	1	27		15 †‡	42 †‡		1 ‡	29 †‡
(c) With Dead Paths	LLVM	58	74	0	2	1442	57 †	73 †‡	1380	0	0
	C11	6	40	0	25		11 †	43 †‡		0	23 †
(d) With Loops	LLVM	49	56	0	0	1763	49 †	56 †	2152	0	0
	C11	6	18	0	7		7 †	20 †		0	10 †
(e) Smaller Tests	LLVM	32	38	0	6	779	32 †	38 †‡	782	0	0
	C11	7	18	5	21		7 †‡	24 †‡		6 ‡	21 †‡

Table 1. Validation results of 100 tests and 11300 steps per class. Erroneous passes: † GVN and ‡ SimplifyCFG.

optimizations took place. Nevertheless, these large examples are not ideal for reporting errors, since in the end-to-end validation, errors in one optimization pass may be masked by a following pass. We therefore also generated some tests with a smaller number of accesses and control paths which reveal the actual bugs. We demonstrate one such set of 100 test cases (e), which revealed the reported bugs #22514 and #22708. Bug #22306 was identified by manual inspection.

Since our validator does not currently handle pointer and array accesses nor loop optimizations such as loop unrolling, we avoided generating programs with such accesses, and generated *do-while* loops, on which the undesired optimizations are not applicable. In principle, the validator could be extended to handle pointer and array accesses and identify such bugs.

Validation Parameters Our validation is parametrized by (*i*) the *LLVM version* tested, (*ii*) the *memory model* (either C11 or the LLVM model), (*iii*) the *validation approaches* (either compiler-independent matching (CIM), or metadata-based (MD)), and (*iv*) the *validation mode* (either end-to-end validation or stepwise validation for the individual transformations). As for the LLVM versions tested, we have chosen LLVM versions 3.6, which was the stable version when the work was done, and version 3.7rc2, a more recent version, in which our reported miscompilation bugs were fixed.

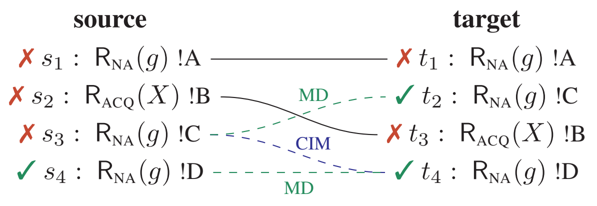
For each test program, we collect the unoptimized IR generated by the `clang++` frontend and optimize it with `opt -O3`. To perform stepwise validation, we used a LLVM command-line parameter that prints the IR before and after every optimization pass. It turns out, however, that the IR after one optimization pass was not always identical to the IR before the next optimization pass. This is because in between LLVM performs various locally scoped optimizations (e.g., within a single basic block), and outputs only the affected IR, which is often difficult to relate to the whole function IR. We therefore ignored these partial IRs; we collected only IRs of the entire function CFG, and apply the validator to both the $\mathsf{IR}^i_{\mathrm{pre}} \rightsquigarrow \mathsf{IR}^i_{\mathrm{post}}$ and the $\mathsf{IR}^i_{\mathrm{post}} \rightsquigarrow \mathsf{IR}^{i+1}_{\mathrm{pre}}$ transformations. We do not validate the IR versions which are same and only validate the non-identical (Non-id) IR pairs. In total, there were 113 such validation pairs per test, only less than 3% of which actually changed the IR.

4.2 Observations

Table 1 reports the results of our experiments. We make the following observations.

First, our CIM validator and the stepwise MD validator are extremely accurate: they report no errors in LLVM 3.7rc2 with respect to the LLVM model, but find plenty of errors in LLVM 3.6, and also some errors in LLVM 3.7rc2 with respect to the C11 model. We note that although many errors have been found, they are often caused by the same compiler bug. For example, all 95 errors found in straightline programs are due to bug #22708.

Second, the metadata-based validator (MD) finds more errors than CIM because it is less prone to having the effect of an invalid optimization being masked by its context. For example, consider the unsafe reordering $\mathsf{R}_{\mathrm{ACQ}}(X); \mathsf{R}_{\mathrm{NA}}(g) \not\rightsquigarrow \mathsf{R}_{\mathrm{NA}}(g); \mathsf{R}_{\mathrm{ACQ}}(X)$ applied to the following program:


source
target
✗ s_1 : $\mathsf{R}_{\mathrm{NA}}(g)$!A
✗ s_2 : $\mathsf{R}_{\mathrm{ACQ}}(X)$!B
✗ s_3 : $\mathsf{R}_{\mathrm{NA}}(g)$!C
✓ s_4 : $\mathsf{R}_{\mathrm{NA}}(g)$!D
✗ t_1 : $\mathsf{R}_{\mathrm{NA}}(g)$!A
✓ t_2 : $\mathsf{R}_{\mathrm{NA}}(g)$!C
✗ t_3 : $\mathsf{R}_{\mathrm{ACQ}}(X)$!B
✓ t_4 : $\mathsf{R}_{\mathrm{NA}}(g)$!D
MD
CIM
MD

CIM matches s_3 with t_4 and considers s_4 is deleted and t_2 is introduced. Since both the deletion and introduction are safe according to the LLVM model, CIM reports no error. MD instead matches s_3 with t_2 and reports an error. For the same reason, smaller tests are sometimes better at exposing errors with the CIM approach.

Third, CIM finds fewer errors end-to-end than with stepwise checking. This is because the effect of an erroneous transformation can be masked by a following transformation. The same, however, does not always hold for MD.

There are several cases, where MD validates all individual steps, but reports and an end-to-end validation error. The following example illustrates those cases:

$$\begin{array}{lclcl}
\textbf{source} & & \textbf{intermed.} & & \textbf{target} \\
\text{if}(*)\{\checkmark \mathsf{R}_{\mathrm{NA}}(g)\ !\mathsf{A}\} & \overset{(1)}{\rightsquigarrow} & \checkmark\ \mathsf{R}_{\mathrm{NA}}(g)\ !\mathsf{A} & \overset{(2)}{\rightsquigarrow} & \checkmark\ \mathsf{R}_{\mathrm{NA}}(g)\ !\mathsf{A} \\
\checkmark \mathsf{R}_{\mathrm{NA}}(g)\ !\mathsf{B} & & \text{✗}\ \mathsf{R}_{\mathrm{NA}}(g)\ !\mathsf{B} & &
\end{array}$$

In the source the accesses are marked with unique metadata nodes which propagate along with the transformations (1) and (2). Both of the $\mathsf{R}_{\mathrm{NA}}(g)$ actions are non-deletable. In the LLVM model the action movement is allowed and thus (1) is correct. Also (2) is correct since $\mathsf{R}_{\mathrm{NA}}(g)!\mathsf{B}$ deletion is safe. However, although both (1) and (2) are safe, the end-to-end $\mathsf{R}_{\mathrm{NA}}(g)!\mathsf{B}$ deletion is reported as a "Possible Error" since MD finds that the non-deletable $\mathsf{R}_{\mathrm{NA}}(g)!\mathsf{B}$ is deleted.

Thus, although MD can be more precise, some of the errors it reports especially in the end-to-end mode are false positives. False positives arise because LLVM occasionally drops or mixes up metadata information, e.g. when creating or merging ϕ nodes, or in cases such as discussed previously. We therefore consider our two validation approaches complimentary: MD is better for validating individual optimizations, whereas CIM is better for end-to-end validation.

In our experiments, we observed that LLVM does not normally perform eliminations and reorderings of atomic accesses. It marks the atomic accesses as *'volatile'* in the IR to avoid any deletion or reordering among them. While this strategy facilitates achieving correctness, various optimization opportunities are lost, which is considered as a potential place for improvement (see LLVM documentation). In contrast, non-atomic shared accesses are heavily optimized (e.g., reordered with atomic accesses and/or deleted).

Out of the roughly 40 LLVM passes applied (some multiple times each), we observed that only 13 actually affected the test programs: SROA, Early CSE, Combine Redundant Instruction, Function Integration/Inline, Reassociate Expression, Dead-Store-Elimination, GVN, Simplify CFG, Jump Threading, SCCP, Value Propagation, LCSSA, and Canonicalize Natural Loop. In two of these passes, we have found errors: GVN and Simplify CFG.

Finally, there are no validation errors according to the C11 model for straightline programs, but several for programs with control flow. This can be explained by observing that these errors arise because of the introduction of speculative memory loads. LLVM, of course, does not introduce loads needlessly: these get introduced as a result of restructuring the program's CFG.

5. Related Work

Compiler correctness is a long-standing research topic and there are various approaches that aim to achieve correct compilation. We categorize them as follows.

Verified Compilation In verified compilation, the compiler comes together with a mechanized proof ensuring that whatever transformations it performs are correct. The most prominent such compiler is CompCert (Leroy 2009), which has also been extended to handle concurrency (Ševčík et al. 2013; Beringer et al. 2014).

Identifying the sound program transformations under a given weak memory model is the first step towards verifying optimizing compilers. Ševčík (2011) first studied this problem in the context of a simple DRF memory model by considering a set of abstract transformations. Later, Morisset et al. (2013) and Vafeiadis et al. (2015) studied the same problem in the context of the C11 memory model. As already discussed, our work builds heavily upon these works.

Translation Validation Translation validation is a simpler verification approach that is typically decoupled from the original compiler development and is reusable for multiple compilers and languages. Given a run of the compiler, it just checks if the target program refines the source program. Since this is undecidable in general, one often relies on clever heuristics (Pnueli et al. 1998; Necula 2000; Tristan et al. 2011) or checks a simpler property that may or may not imply refinement. An alternative scheme is to instrument the compiler to augment program to facilitate the validation (Namjoshi and Zuck 2013). The MD validator follows this scheme and instruments LLVM. However, compared to Namjoshi and Zuck (2013), the instrumentation effort and the extracted information is significantly less in our MD validation and is easily replicable across compilers.

Prior to this work, no validation work for C11 concurrency compilation existed. Our approaches can be seen as translation validations with the crucial difference that we only check that the memory access sequences in the two programs correctly match up and not program equivalence or refinement. We catch the concurrency-related errors which are not identifiable by the existing sequential validators.

Compiler Testing Another approach for improving compiler trustworthiness is extensive testing. Here, many automatically generated test programs are compiled with and without optimizations, executed, and their results are compared to check for optimization errors. Although testing does not ensure correctness, it has been extremely effective at finding bugs (Yang et al. 2011; Le et al. 2014).

Morisset et al. (2013) have applied testing to check for C11 concurrency errors in GCC. They instrument the compiled programs so as to record the sequence of memory accesses performed, and then try to match the sequence of accesses among the two versions of the program (with and without optimization).

Our approach is closely related to that of Morisset et al. (2013), but has important differences. The major one is that we compare two C11 program CFG structure, whereas Morisset et al. (2013) compare two particular executions. Thus, our matching algorithms are sufficiently more complicated because it considers the programs' CFGs, which may be structurally dissimilar because of transformations.

A second difference is that we perform validation at the compiler IR level, whereas Morisset et al. (2013) do it at the assembly level. Matching at the assembly code is problematic because the conversion to assembly loses a lot of the information present at the IR level, such as the memory order annotations. For example, consider the two transformations:

$$\mathsf{W}_{\mathrm{RLX}}(X);\mathsf{W}_{\mathrm{RLX}}(Y) \rightsquigarrow \mathsf{W}_{\mathrm{RLX}}(Y);\mathsf{W}_{\mathrm{RLX}}(X)$$
$$\mathsf{W}_{\mathrm{REL}}(X);\mathsf{W}_{\mathrm{REL}}(Y) \rightsquigarrow \mathsf{W}_{\mathrm{REL}}(Y);\mathsf{W}_{\mathrm{REL}}(X)$$

The first transformation is valid, whereas the second one is not. At the assembly level, however, the $\mathsf{W}_{\mathrm{REL}}$ and $\mathsf{W}_{\mathrm{RLX}}$ events are indistinguishable: they are both `MOV` instructions. Thus, by performing matching just at the assembly level, one will necessarily miss a number of bugs or will report many false positives. Matching at the IR level enables us to provide better precision and cover a broader set of transformations.

Recently, Chong et al. (2015) have also used testing approaches to check OpenCL compilers and have found over 50 bugs in commercial compilers. Although these bugs were exposed by compiling concurrent programs, manually reducing the test cases revealed that none of the bugs found were actually inherently concurrency-related.

6. Conclusion

We developed a technique for validating LLVM optimizations with respect to concurrency. Our validator has proved useful in finding concurrency-related compiler bugs, and could in principle be integrated in the compiler's regression testing suite. Nevertheless, doing so in a useful fashion would require more implementation work. In particular, one would need to extend the validation to handle more LLVM features, such as mixed sized accesses, and to come up with good heuristics for dealing with loop optimizations.

In the future, we intend to pursue a more theoretical line of work: to develop a formal definition of the LLVM memory model and to prove that the expected transformations are sound according to it.

Acknowledgments

We would like to thank Marko Doko, Ori Lahav, and the anonymous reviewers for their comments that improved the content of this paper. The research was partially supported by the EC FP7 project ADVENT.

References

M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL'11*, pages 55–66. ACM, 2011.

M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In *POPL'12*, pages 509–520. ACM, 2012.

L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory C. In Z. Shao, editor, *ESOP 2014*, volume 8410 of *LNCS*, pages 107–127. Springer, 2014.

N. Chong, A. F. Donaldson, A. Lascu, and C. Lidbury. Many-core compiler fuzzing. In *PLDI'15*. ACM, 2015.

R. Elhorst. Lowering C11 atomics for ARM in LLVM. In *European LLVM Conference*, 2014.

ISO/IEC 14882:2011. Programming language C++.

ISO/IEC 9899:2011. Programming language C.

V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *PLDI'14*, pages 216–226. ACM, 2014.

X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

LLVM documentation. LLVM atomic instructions and concurrency guide. `http://llvm.org/docs/Atomics.html`.

J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL'05*, pages 378–391. ACM, 2005.

R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI'13*, pages 187–196. ACM, 2013.

K. S. Namjoshi and L. D. Zuck. Witnessing program transformations. In *SAS*, pages 304–323, 2013.

G. C. Necula. Translation validation for an optimizing compiler. In *PLDI'00*, pages 83–94. ACM, 2000.

A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *TACAS'98*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.

S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *PLDI'12*, pages 311–322. ACM, 2012.

J. Ševčík. Safe optimisations for shared-memory concurrent programs. In *PLDI'11*, pages 306–316. ACM, 2011.

J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, June 2013.

R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken. Data-driven equivalence checking. In *OOPSLA'13*, pages 391–406. ACM, 2013.

J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI'11*, pages 295–305. ACM, 2011.

V. Vafeiadis and F. Zappa Nardelli. Verifying fence elimination optimisations. In *SAS'11*, volume 6887 of *LNCS*, pages 146–162. Springer, 2011.

V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL'15*, pages 209–220. ACM, 2015.

X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI'11*, pages 283–294. ACM, 2011.