Validating Optimizations of Concurrent C/C++ Programs

Soham Chakraborty and Viktor Vafeiadis

Presented by
Akshay Gopalakrishnan

August 26, 2021

Introduction

- Lot of optimizations done at the LLVM IR level for C/C11 programs.
- However, very conservative when it comes to transformations involving shared memory accesses.
- Prior work formalized a major fragment of the LLVM memory model.
- This helped to prove that many transformations not done were infact safe to do.
- The current work introduces a validator that verifies whether LLVM transformations are safe to do wrt the C11 and LLVM memory model.

Simple example

```
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.

C11/LLVM memory model: Common elements

Partial orders on program elements

- Program order.
- Syncrhonize with order.
- Reads-from.

Access orders:

- Non-Atomic - default access mode of shared memory.
- Release - for write accesses.
- Acquire - for read accesses.
- Sequentially consistent - default atomic access mode for shared memory.

C11/LLVM memory model: Example showing their difference

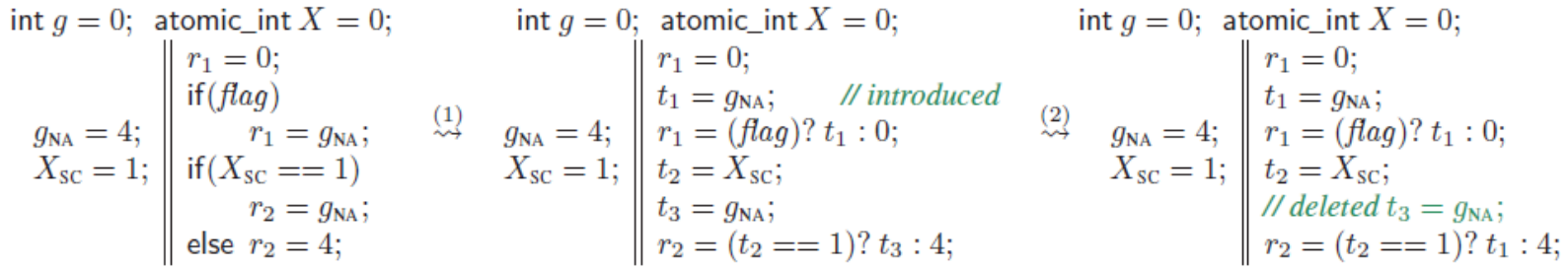


Figure 3. A sequence of LLVM transformations: (1) introduce a speculative read of g during CFG simplification, (2) remove redundant read of g by the GVN pass. The composition violates the "roach motel" property when $flag = false$.

Explanation

- The first transformation reorders the read of g_{NA} to be before the conditional.
- This transformation is invalid under C11 memory model, but is extensively done in LLVM.
- The reason for this is the second transformation which does Global Value Numbering (a form of redundant code elimination).
- These two transformations in unison prevent multiple reads from g_{NA}.
- While it is safe to perform the second transformation in C11, the first violates the std "roach motel" reordering constraint, thus giving us a new outcome for when flag is set to false.

Other errors due to transformations

$$
\begin{array}{l}
\textbf{for}\,(i = 0\,;\, i < 4\,;\\
\quad i{+}{+})\{\\
\quad g[i]_{\text{NA}} = 0;\\
\quad X[i]_{\text{SC}} = i;\\
\}
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{l}
X[0]_{\text{SC}} = 0;\\
X[1]_{\text{SC}} = 1;\\
X[2]_{\text{SC}} = 2;\\
g[0:3]_{\text{NA}} = 0;\\
X[3]_{\text{SC}} = 3;
\end{array}
$$

Context:

$$
\left[\; - \;\middle\|\; \begin{array}{l} \textbf{if}\,(X[2]_{\text{SC}})\\ \quad r = g[2]_{\text{NA}}; \end{array} \right]
$$

Figure 4. SLP vectorizer performs an unsafe reordering.

$$
\begin{array}{l}
\textbf{if}\,(\mathit{flag})\,\{\\
\quad g_{\text{NA}} = 5;\\
\}\\
r = X_{\text{ACQ}};\\
r' = (r\;?\;g_{\text{NA}} : 8);
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{l}
\textbf{if}\,(\mathit{flag})\,\{\\
\quad g_{\text{NA}} = 5;\\
\quad t = 5;\\
\}\;\textbf{else}\,\{\,t = g_{\text{NA}};\,\}\\
r = X_{\text{ACQ}};\\
r' = (r\;?\;t : 8);
\end{array}
$$

Context:

$$
\left[\; - \;\middle\|\; \begin{array}{l} g_{\text{NA}} = 8;\\ X_{\text{REL}} = 1; \end{array} \right]
$$

Figure 5. GVN performs an unsafe reordering.

Validating Optimizations of Concurrent C/C++ Programs

Other errors due to transformations

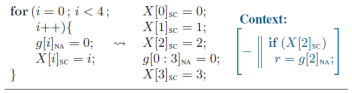


Figure 4. SLP vectorizer performs an unsafe reordering.

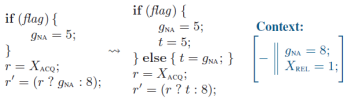


Figure 5. GVN performs an unsafe reordering.

Both these examples exhibit unsafe reordering. It is interesting to note that several complex transformations involve this basic transformation at its elementary level.

Validation Approaches

Two kinds:

- Compiler independent matching - No IR or Language specific constructs required (only memory model suffices).
- LLVM specific matching - Makes use of IR metadata to improvise validation.

Three possible results of matching:

- Correct - transformation is safe and valid.
- Possible error - transformation is unsafe (one of them).
- Unknown - cannot decisively conclude on the validity (due to loops).

Compiler Independent Mathcing (CIM)

- Consider a source program and a target program after performing a set of transformations.
- Mark the events in the source and target program using some tags (next slide).
- Perform matching between events of source and that of target.
- Matchings misaligned (cross matchings) imply reordering applied.
- No matching for target event implies Introduction or speculative transformation applied.
- No matching for source event implies elimination transformation applied.
- Analyze based on this information.

CIM: Basic Tags

- Non-deletable - Cannot be eliminated.
- Conditionally deletable - Can be eliminated subject to another transformation being applied to another event in the same thread.
- Immediately Deletable - Can be eliminated right away.

CIM: Basic Tags

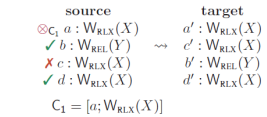
- Non-deletable - Cannot be eliminated.
- Conditionally deletable - Can be eliminated subject to another transformation being applied to another event in the same thread.
- Immediately Deletable - Can be eliminated right away.

The conditionally deletable is the most tricky part. It relies on some other transformation to have been done in the source, which is something in my eyes very difficult to assert. Firstly, which transformation and secondly on which events. The lack of any proofs given based on their assignment of tagging makes it a little more difficult to have intuition on this tag been given to events. They have covered some cases on when to conditionally match, but is it sound ? Also, is it complete? Also, note that these tags do not help us decide reordering or Introduction in any way.

CIM: Example using tags

source		target
$\otimes_{\mathsf{C}_1}\ a : \mathsf{W}_{\mathrm{RLX}}(X)$		$a' : \mathsf{W}_{\mathrm{RLX}}(X)$
✓ $b : \mathsf{W}_{\mathrm{REL}}(Y)$	$\rightsquigarrow$	$c' : \mathsf{W}_{\mathrm{RLX}}(X)$
✗ $c : \mathsf{W}_{\mathrm{RLX}}(X)$		$b' : \mathsf{W}_{\mathrm{REL}}(Y)$
✓ $d : \mathsf{W}_{\mathrm{RLX}}(X)$		$d' : \mathsf{W}_{\mathrm{RLX}}(X)$

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



Note here that the release write to Y is non-deletable (as deleting it would result in syncrhonization errors). Additionally, the last write to X is non-deletable (elminiation would result in unsafe behaviors such as reading a previous write not supposed to be visible.) The third write is deletable because a write proceeding it replaces its value (also it is relaxed order and a proof for soundness of such a transformation exists). The first one is conditional on the third write being reordered before the release write (as shown in right hand side of figure). Note that tagging of conditionally deletable is non-trivial.

CIM: How to mark accesses using tags?

Initially, we mark all actions non-deletable and then go on to re-mark them as deletable or conditionally deletable. The latter tag is non-trivial to assign, here are a few observations on them:

- Insufficiency of release-acquire pairs: Not sufficient to ensure an event is non-deletable
- C11 release sequences: Certain release accesses can also be deleted (if strengthening transformation is performed)
- Synchronization access deletion: Although allowed under certain circumstances (low level Fence eliminations), it is avoided at all costs (goes against programmer intention and may cause deadlock).

Validating Optimizations of Concurrent C/C++ Programs

CIM: How to mark accesses using tags?

CIM: How to mark accesses using tags?

Initially, we mark all actions non-deletable and then go on to re-mark them as deletable or conditionally deletable. The latter tag is non-trivial to assign, here are a few observations on them:

- Insufficiency of release-acquire pairs: Not sufficient to ensure an event is non-deletable
- C11 release sequences: Certain release accesses can also be deleted (if strengthening transformation is performed)
- Synchronization access deletion: Although allowed under certain circumstances (low level Fence eliminations), it is avoided at all costs (goes against programmer intention and may cause deadlock).

The release-acquire pair Insufficiency is validated by an example that reorders a release with a subsequent acquire. I remember in our analysis for JavaScript, we deemed such a reordering to be invalid as it introduces happens-before cycles. More importantly, our reasoning was that this would go against programmer intuition as the synchronizations placed by them should ideally work, and must not suddenly stop due to some addition of syncrhonization by the compiler. However, this actually means that a new behavior can be formed. This still has to be proved.

CIM: Matching source with target

Once all the marking of events are done, matching follows. Here are the steps taken:

- Match all Synchronization events first. If a mismatch is found, return "Possible Error".
- Now for each non-deletable source action not matched, define a window between events that this event cannot be reordered out of. Now match these corner events with target. This window will be our place to search for a match. If a match is not found, it certainly implies an incorrect reordering has been done. Thus we can return Possible Error.
- Now for all deletable target actions, we perform the same matching as that of the previous. If a match is not found, we consider it as introduction transformation and analyze further.

CIM: Matching Source with Target cnt'd

Any unmatched target action implies introduction transformation.

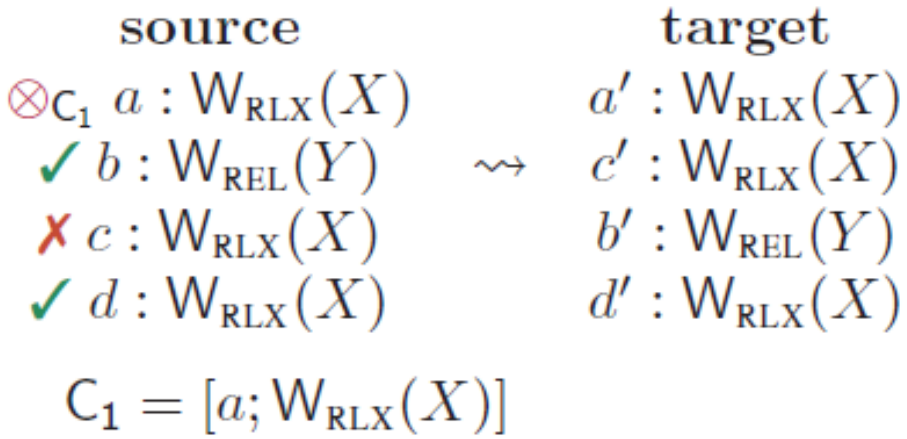
- Write/Updates: Atomic is disallowed (Possible Error) but certain cases of non-atomic allowed (Correct).
- Reads: Incorrect as per C11 but safe as per LLVM model (Unknown).
- Fences: Just adds synchronization so safe (Correct).

Any unmatched target action implies introduction transformation.

- Write/Updates: Atomic is disallowed (Possible Error) but certain cases of non-atomic allowed (Correct).
- Reads: Incorrect as per C11 but safe as per LLVM model (Unknown).
- Fences: Just adds synchronization so safe (Correct).

Fence introduction, as per my work on JavaScript may be incorrect as if we prove transformations by preserving existing happens-before relations, then this would introduce a cycle. This cycle would be illegal. Though one may not observe this cycle in practice. This is something to note/observe in our future endeavors tackling transformations.

CIM: Example Matching

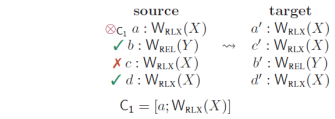


Note: $W_{REL}(X)$ should be non-deletable.

Validating Optimizations of Concurrent C/C++ Programs

CIM: Example Matching

CIM: Example Matching



Note: $W_{REL}(X)$ should be non-deletable.

The numbers in the matching lines indicate the step number as per previous slide.

CIM: Moving towards Conditional Branching

- For each target code path, identify a set of source paths.
- Apply matching analysis as above for each such pair.
- The target program is safe if all such possible mappings are Correct.

LLVM Specific Matching using MetaData (MD)

- LLVM has a feature of attaching metadata to programs that is preserved throughout the pipeline of transformations.
- This feature is used to attach information about events and relations to any future events.
- This information is used to decide whether transformations performed were safe overall.
- Examples of such information added are: OrderedPairs(a,b), PathConditions(a), etc.

Observations on Testing both Methods: CIM vs MD

- CIM and MD are extremely accurate (hence the Bugs they found).
- CIM misses some errors due to mismatching events and taking some events to be introduced and some eliminated.
- MD finds more errors than CIM.
- MD has false positives - Due to end-to-end validation error (only see source target, but no intermediate steps).
- CIM is better for end-to-end validation.
- MD is better for individual step-wise validation.

Observations on Testing both Methods: CIM vs MD

- CIM and MD are extremely accurate (hence the Bugs they found).
- CIM misses some errors due to mismatching events and taking some events to be introduced and some eliminated.
- MD finds more errors than CIM.
- MD has false positives - Due to end-to-end validation error (only see source target, but no intermediate steps).
- CIM is better for end-to-end validation.
- MD is better for individual step-wise validation.

LLVM keeps playing with metadata, sometimes even discards some. This might also be the reason for MD to be having False Positives. However, the example the authors used to show MD has false positives seems to show that CIM will also give the same outcome. But, this is not the case because the event in target is A, for whcih the only path we get from source is the conditional fails. This tells me that they do not particularly work at the value level. This might become problematic as I am unsure how they tackle it.

Thank you

Questions?