Building a Compiler Optimizing C++ Atomic Accesses

July, 2023

Zhiyang Liu

Building a Compiler Optimizing C++ Atomic Accesses

THESIS

by

Zhiyang Liu

to obtain the degree of Master of Science
at Delft University of Technology

Student id: 5534496
Email: z.liu-57@student.tudelft.nl

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Daily Supervisor: Dr. S. Chakraborty, Faculty EEMCS, TU Delft
Committee Member: Dr. Z. Al-Ars, Faculty EEMCS, TU Delft

Abstract

Atomics is an important primitive for programming languages like C++ to develop concurrent software. Atomic variables, together with weak memory models allow for a bigger space for instruction reordering and compiler optimizations. However, the current compilers like LLVM do not support many transformations of atomics, which may lose chances of optimizations.

In this thesis project, we built a compiler that optimizes C++ atomic memory accesses based on LLVM 14.0.0. We modified related LLVM passes to enable these optimizations. Specifically, our compiler is able to optimize Read-After-Read (RAR), Read-After-Write (RAW), and Overwritten Write (OW) patterns containing atomics. To achieve this, we removed checks in LLVM that forbid atomic accesses from being processed. And we added constraints and adapted them into existing algorithms of LLVM passes, to ensure the soundness of our transformations.

We tested our compiler using randomly generated ordered memory accesses. And our compiler is shown to be able to eliminate 2% - 15% redundant atomic instructions in our test sets, which the current LLVM cannot optimize, while hardly hurting the compile time (less than 1%). And we evaluated our compiler using several concurrent applications. We have not yet found a significant performance gain after building these applications using our compiler. The reason could be that these concurrent benchmarks do not contain the patterns our compiler optimizes.

Preface

Looking back at the time when I reached Delft for the first time, I could not imagine I am now working on compilers and finally finishing this thesis project. When I first stepped into the lecture hall of the first compiler course totally out of interest, I knew literally nothing about compiling technology. I barely acquired any programming skills at that time even. It was the help, support, and inspiration from all the kind people by my side that enabled me to reach here.

As my daily supervisor, Soham is really patient and nice to me. Without his generous assistance, it could be infeasible for a rookie to start a journey in the LLVM project. It was him that provided me with this chance to work on a product compiler. The experience with LLVM is wonderful and learned me a great lesson in compiler structures, and even program design.

I also extend my thanks and condolences to Eelco. It was this brilliant, interesting gentleman that first raised my interest in this totally unfamiliar field. I can still remember the excitement when I built a working compiler front end. Even just before his untimely passing, I received an email from him inviting students to take his programming language project course. His spirit will always encourage me.

And I am grateful for all the other professors, friends in the PL group. This is a precious memory for me. I am just new to the game, and it is their passion and friendliness that helped me keep going.

And additional thanks go to my parents and my girlfriend, who have been continuously supporting me even if they have little idea about my work. My faith in getting back to you and starting my new journey with you is my spiritual backbone in those tough periods.

Finally, I would invite any interested people, even beginners who know nothing like I once was, to try the programming language and compiler courses. The people of the group and all the harvest along the way will not let you down.

Zhiyang Liu
Delft, the Netherlands
July, 2023

Contents

1 Introduction

1.1 Context

A compiler works as a bridge between programmers and machines. It translates the source programming language to the target language. And compilers have always been beyond translators. Smart modern compilers perform a tremendous number of aggressive optimizations on the program to obtain better performance and expected code size. On the other hand, programming languages do not stay still. With the development of multi-core processors and parallel computing, programming languages have been gifted new features to adapt.

C/C++ 11 introduced atomic types to support atomic memory accesses. Different threads can simultaneously operate on the same atomic object without causing a data race. This new feature enables programmers to conveniently write lock-free concurrent codes. However, as relatively new guests to compilers, atomic accesses are not well served. On LLVM, a state-of-the-art compiler frame, atomic accesses are treated conservatively. A number of normal optimizations are not performed on these accesses, such as common sub-expression elimination (CSE) and Dead Store Elimination (DSE). Instead, they are dealt with nearly the same as volatile operations, kept almost intact in output codes.

Such conservative behaviors are understandable. Because compilers must not miscompile. As a basic part of the infrastructure in the world of computer science, an erroneous compiler could be catastrophic. Without careful study and verification, aggressive optimizations on atomic accesses may cause mistakes. Keeping these newcomers as they are is no fault.

In this thesis, based on reordering rules found in related work [17], we attempt to enable a group of feasible optimizations on atomic accesses. We implemented these optimizations on LLVM optimizer. As a widely used product-quality compiler, LLVM provides various optimizations on memory accesses. And for atomic accesses, LLVM simply leaves them as is. Developers of LLVM carefully rule out atomic accesses in optimizing phases. And without these annoying ordered accesses, optimizations stay valid. We implemented our optimizations based on the existing LLVM optimizer. Users of this work can enable these optimizations using some simple flags.

1.2 Problem Statement

This thesis project aims to enable optimizations on C/C++ 11 atomic memory accesses and implement these optimizations on LLVM. To achieve this, we may need to answer the research questions below:

- What does the existing LLVM do with memory accesses? Why does not it optimize atomic accesses? How does it treat atomic accesses?
- What optimizations are feasible for atomic accesses?
- How to enable such optimizations on LLVM? Will such modifications hurt the performance of the compiler?
- How much can we reduce the number of lines of the programs?

1.3 Contributions

Contributions of this work include:

- We implemented optimizations for atomic memory accesses on LLVM, which LLVM does not support yet.

- Though our work targets C++ atomics, it can be applied to other source languages as long as they share similar memory models. Our work is on the optimizer phase of LLVM, which means these optimizations are target-independent and source-independent.
- As an experimental work to modify LLVM, we found problems with the current LLVM. Atomics are carefully ruled out from core libraries to optimizing passes. If LLVM will support optimizations for atomics in the future, it may need to modify its alias analysis, IR properties, and pass implementations.
- We tested our work using randomly generated memory accesses. And we showed our compiler is able to eliminate redundant memory accesses, which the current LLVM does not do.
- We also evaluated our work using some concurrent applications. We have not yet observed significant performance gain. Nevertheless, it is possible to find performance improvements when a potential benchmark is found, or when more aggressive optimizations are done.

1.4 Outline

The remainder of this thesis is organized as follows: Section 2 gives the necessary background information about C++ atomics, LLVM compiler, and possible transformations of atomics. In Section 3 to Section 6, we describe details of the problem together with solutions proposed by us. Our work focuses on three passes in LLVM: EarlyCSE, InstCombine, and DSE, which are discussed in respective sections. In Section 4, we did a summary of works in Section 3, which may help in understanding the following sections. Section 7 provides our detailed experimental evaluation and results. Related and future work are discussed in Section 8. Finally, we conclude our thesis in Section 9.

2 Background

In this section, we give some necessary background information for understanding subsequent sections. First, we introduce LLVM, a state-of-art compiler framework, on which our work is based. And after that, we provide the theoretical basis for the optimizations to be implemented.

2.1 C++ Atomics and Memory Orders

First of all, we briefly introduce the C++ atomic and memory orders. Atomic variables were originally created to manage access to shared memory in multi-thread areas. As a typical usage, one thread may produce data and writes to an atomic variable. Then another thread may read this atomic variable. If it gets the expected value, it will use the data created by the first thread.

Each atomic memory access has a memory order. It specifies how memory accesses are to be ordered around an atomic operation. By default, the memory order is `std::memory_order_seq_cst`. And different memory orders have the following restrictions:

- **memory_order_relaxed** As the name indicates, relaxed operations have no constraints for their ordering. They do not synchronize with other operations. They are only guaranteed not to raise a data race.
- **memory_order_acquire** A load operation with acquire order requires that no reads or writes in the current thread can be reordered before it. Also, all release writes in other threads on the same atomic variable are visible in the current thread.
- **memory_order_release** A store operation with release order requires that no reads or writes in the current thread can be reordered after it. And all writes in the current thread are visible in other threads that acquire the same atomic variable.
- **memory_order_acq_rel** Combination of acquire and release.
- **memory_order_seq_cst** Besides the ordering enforced by acquire and release, sequentially consistent ordering also requires that all threads observe all modifications in the same order.

An example of using C++ atomics is shown below. And programmer can explicitly specify the memory order to use by passing it as a parameter of the atomic operations.

```
#include<atomic>
using namespace std;
...
// Non-atomic version:
int a, b;
a = b;

// Atomic version:
atomic<int> a, b;
atomic_store(&a, atomic_load(&b));
// As a simpler way, one can also write:
a.store(b.load());
```

There are many models of memory. And by specifying memory orders of memory accesses, the programmer can choose the model to use. When the memory model is omitted like the program above, the default model will be

sequentially consistent. It has similar restrictions to sequential programs on reordering memory accesses. The major difference is that it also has inter-thread constraints. See this example:

$$\begin{array}{l} Y = 2; \\ X = 1; \end{array} \Bigg\| \begin{array}{l} if(X == 1) \\ assert(Y == 2); \end{array} \qquad \text{(EX\_SC)}$$

Here we can see, X and Y are unrelated variables. However, the assertion cannot fail. Because a sequentially consistent model requires a consistent total order of memory accesses. If `X==1` reads the value 1 written by `X=1`, then it must also see operations before this write. Then the read of `Y` must get the value `2` written before the store of `X`.

From the view of a compiler, memory models confine the space of optimization. A weaker model gives many possibilities for optimization. Since relaxed accesses do not synchronize with others, they can be reordered relatively freely. And the sequentially consistent model is the most expensive one. It enforces strict ordering on all accesses. Practically, this means losing most opportunities for optimizations. And an acquire-release model is somewhere in between. It disallows moving stores and loads across respective atomic operations, leaving some space for optimizations.

2.2 The LLVM Compiler Infrastructure

LLVM is a collection of compiler and toolchain technologies [12]. It follows the classical three-phase compiler design. In an LLVM-based compiler, there are three basic parts as shown in Figure 1 [7]:

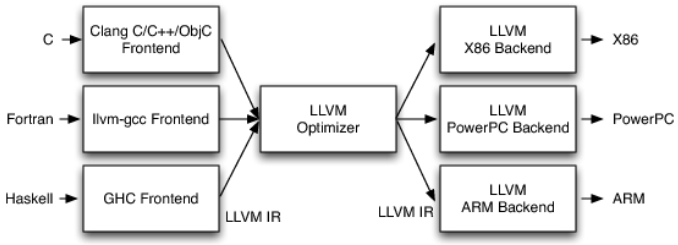


Figure 1: Typical Design of an LLVM-based Compiler [7]

- **Front end**. A front end parses, validates and throws errors in the input code. After these checks, It translates the source code into LLVM Intermediate Representation (LLVM IR), a language-independent intermediate representation.
- **Optimizer**. In this phase, various optimizations take place. And the code is improved.
- **Back end**. After being optimized, the IR code is finally fed into a code generator to produce the target code.

This thesis work is built on the optimizer phase. The optimizer consists of a set of passes. Basically, there are two important types of passes in LLVM Optimizer: Analysis passes and transform passes. Analysis passes do not modify the input IR code. They collect information that other passes may utilize or for the purpose of debugging. Transform passes, on the other hand, mutate the program, and (usually) improve it in some way. And each (transform) pass performs a family of optimizations. For example, Dead Store Elimination (DSE) pass focuses on finding and removing dead writes.

For our project, we need to investigate how the current LLVM optimizes memory accesses and how atomic accesses and non-atomic accesses are treated differently.

Our work uses LLVM 14.0.0, which was the newest version of LLVM when the project started. At the time this thesis is written, the most recent release is LLVM 16.0.0. Nevertheless, the passes where we made modifications did not change significantly. And the problem that atomic accesses cannot be optimized remains there. So the version of LLVM will not be a problem here.

2.3 Current LLVM Approach to Atomic Operations

To observe the existing LLVM behavior with atomic accesses, first consider this example:

```
1 std::atomic<int> X;
2 int a;
3 int main(){
4     a = 1;
5     X.store(2, std::memory_order_relaxed);
6     a = 3;
7 }
```

Obviously, the write `a = 1` on line 4 is dead, since it is overwritten by line 6. And the compiler should remove the first write as an optimization. After parsing and optimizing this program using `clang++ -O3`, we check the output IR (LLVM Intermediate Representation) file shown below. Note that atomic variables in C/C++ 11 are implemented using `struct`, hence reads and writes on them will use the `getelementptr` instruction to access the boxed value.

```
1 define i32 @main() {
2 entry:
3   store atomic i32 2, i32* getelementptr inbounds (%"struct.std::atomic", %"
        struct.std::atomic"* @X, i64 0, i32 0, i32 0) monotonic, align 4
4   store i32 3, i32* @a, align 4, !tbaa !3
5   ret i32 0
6 }
```

We can see that the compiler did its job. The redundant store with value 1 is removed. This is called dead store elimination (DSE). When the compiler sees the store `a = 3;`, it reversely looks for stores overwritten by this one. The relaxed atomic store of `X` is in its way but is not a clobber. An instruction would be a clobber if we cannot reorder another instruction with it according to the rules in Table 1. Here, neither is the store of `X` a store to location `a` nor is it a strongly ordered access preventing reordering. However, if we replace this program with the following similar version, and keep the same optimization level:

```
1 std::atomic<int> X;
2 int a;
3 int main(){
4     X.store(1, std::memory_order_relaxed);
5     a = 2;
6     X.store(3, std::memory_order_relaxed);
7 }
```

In this example, we simply exchanged `a` and `X`. The first store to `X` should be killed by the write on line 6 for the same reason as the previous example. But the output becomes the following:

```
define i32 @main() {
entry:
  store atomic i32 1, i32* getelementptr inbounds (%"struct.std::atomic", %"
      struct.std::atomic"* @X, i64 0, i32 0, i32 0) monotonic, align 4
  store i32 2, i32* @a, align 4, !tbaa !3
  store atomic i32 3, i32* getelementptr inbounds (%"struct.std::atomic", %"
      struct.std::atomic"* @X, i64 0, i32 0, i32 0) monotonic, align 4
  ret i32 0
}
```

Now the redundant store, which should have been removed, remains there. And if we change the first example in another way, by replacing the access in the middle with an acquire read:

```
std::atomic<int> X;
int a;
int main(){
    a = 1;
    X.load(std::memory_order_acquire);
    a = 3;
}
```

And check the output:

```
define i32 @main() {
entry:
  store i32 1, i32* @a, align 4, !tbaa !3
  %0 = load atomic i32, i32* getelementptr inbounds (%"struct.std::atomic", %"
      struct.std::atomic"* @X, i64 0, i32 0, i32 0) acquire, align 4
  store i32 3, i32* @a, align 4, !tbaa !3
  ret i32 0
}
```

Again, the first store stays there. In this case, the acquire read of `X` works as clobber in the current LLVM, though it should not be.

Similar problems happen to several different types of optimizations in LLVM. This is basically because LLVM treats shared atomic variables conservatively. And from the examples shown above, the problems come in two aspects:

- **Atomics are not removable.** In an optimizing pass, when the iterator meets atomic access, it simply skips the access. As a result, all atomic accesses will remain in the output IR file, regardless of whether they should be deleted.
- **Reordering is disallowed.** Only a limited portion of reordering is actually done in the current LLVM. As shown in the examples, the compiler can reorder a non-atomic access with a relaxed atomic access, but not with an acquire atomic access. Though this transformation should be allowed.

We will discuss the specific problems in more detail in subsequent sections.

2.4 Possible Transformations of Atomic Accesses

C/C++ defines memory models and atomic accesses with memory orders to provide different levels of constraints, and hence different spaces for optimizations. Knowledgeable concurrent programmers should make good use of memory models to write efficient software. But now there is a gap between the transformations allowed by memory models and the actually implemented optimizations implemented by the compiler.

In this project, we mainly consider the elimination optimizations. They are also main optimizations performed on regular memory accesses. First, we consider the cases where one of two adjacent memory accesses kills the other. There are three typical patterns to be optimized, and the respective rules are listed below [4]:

- **Overwritten Write (OW)** If two adjacent stores write to the same location, and the second store has a stronger or equal order, the first store can be deleted:
$$St_{o'}(l, v'); St_o(l, v) \rightsquigarrow St_{o'}(l, v), \textit{when } o' \sqsubseteq o$$
- **Read After Write(RAW)** If a load immediately follows a store that writes to the same location, and the store has a stronger or equal order, the load can be deleted:
$$St_o(l, v); Ld_{o'}(l) \rightsquigarrow St_o(l, v), \textit{when } o' \sqsubseteq o$$
- **Read After Read (RAR)** If two adjacent loads read from the same location, and the first load has a stronger or equal order, the second load can be deleted:
$$Ld_o(l); Ld_{o'}(l) \rightsquigarrow Ld_o(l), \textit{when } o' \sqsubseteq o$$

These rules provide additional constraints that the compiler should obey when eliminating dead instructions. But in practice, the dead instruction and the killing instruction are usually not adjacent. So in order to optimize such patterns, the compiler first needs to reorder the instructions. And the elimination optimizations can only happen if such reordering is allowed:

$$St_{o'}(l, v'); C; St_o(l, v) \rightsquigarrow C; St_{o'}(l, v'); St_o(l, v)$$
$$St_o(l, v); C; Ld_{o'}(l) \rightsquigarrow St_o(l, v); Ld_{o'}(l); C$$
$$Ld_o(l); C; Ld_{o'}(l) \rightsquigarrow Ld_o(l); Ld_{o'}(l); C$$

where reordering every instruction in C with the instruction to be deleted is allowed.

Fortunately, it has been proved that some types of reordering of atomic accesses are legal, and may thus enable optimizations [17]. Table 1 shows the reordering rules of atomic accesses.

$\downarrow a \backslash b \rightarrow$	$R_{NA\|RLX\|ACQ}(l')$	$R_{SC}(l')$	$W_{NA}(l')$	$W_{RLX}(l')$	$W_{REL\|SC}(l')$
$R_{NA}(l)$	✓	✓	✓	✓	✗
$R_{RLX}(l)$	✓	✓	✓	✓	✗
$R_{ACQ\|SC}(l)$	✗	✗	✗	✗	✗
$W_{NA\|RLX\|REL}(l)$	✓	✓	✓	✓	✗
$W_{SC}(l)$	✓	✗	✓	✓	✗

Table 1: Allowed Reorderings $a; b \rightsquigarrow b; a$, assuming locations $l \neq l'$

This table summarizes the rules we need to obey when reordering and optimizing atomic accesses. With this in hand, we can now step into LLVM, and look for opportunities to improve.

3 EarlyCSE Pass

3.1 Overview EarlyCSE

3.1.1 Common Subexpression Elimination

CSE stands for common sub-expression elimination. For example, in the following code:

```
a = b + c * d;
e = f - c * d;
```

`c * d` is computed twice as a common sub-expression. As an optimization, we can compute it only once, and reuse the value:

```
tmp = c * d;
a = b + tmp;
e = f - tmp;
```

Such optimizations are called common sub-expression elimination (CSE). In practice, simple cases as shown in the example above are trial enough for source program developers to find and avoid. The real major source for CSE is the intermediate code generated by compilers, such as accessing data arrays. In LLVM, EarlyCSE is designed to remove trivially redundant instructions. It is called "early" because typically in the sequence of optimizing passes, EarlyCSE pass is invoked before other passes. It aims to find trivial cases and remove instructions efficiently so that other passes (usually with higher complexity) spend less time looking for these simple patterns. All three patterns mentioned in Section 2.4 are targets of EarlyCSE and will be optimized by it.

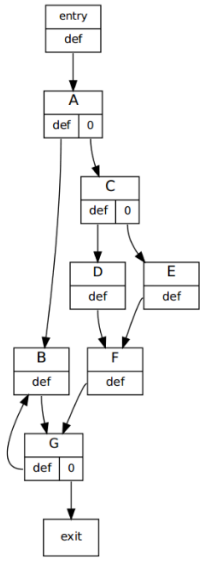
3.1.2 Dominator Tree in LLVM

Consider the following dummy program with basic blocks named from A to G:

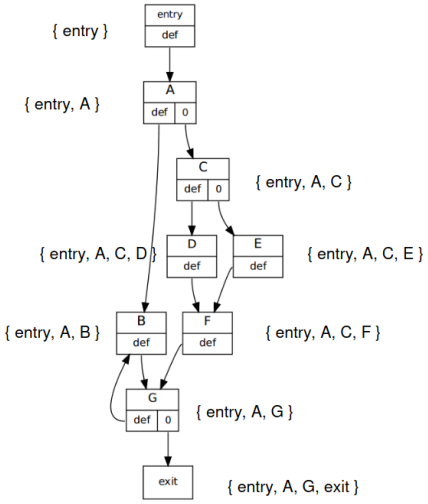
```
entry:
A:
    a = 1
    if(b<2) goto B
C:
    c = 3
    if(d<4) goto D
E:
    e = 5
    goto F
D:
    e = 6
F:
    f = e
    goto G
B:
    g = a
G:
    h = 7
    if(i<8) goto B
```

Listing 1: A Program with Control Flow

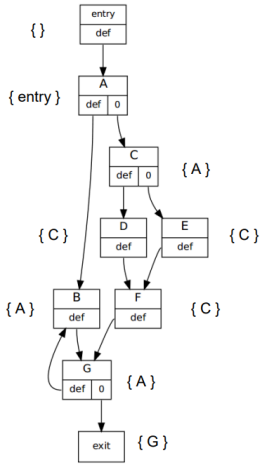
Based on the control flow of the program, we can build a control flow graph (CFG) as shown in Figure 2a. And we define dominance relation as Node X dominates node Y, if and only if all paths from entry to X go through Y. Based on this definition, we can list dominators of each node as shown in Figure 2b.



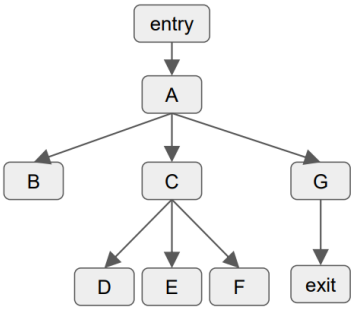
(a) Control Flow Graph



(b) Dominators of Each Node



(c) Immediate Dominators



(d) Dominator Tree

Figure 2: From CFG to Dominator Tree [6]

We pick the immediate dominator of each node as shown in Figure 2c. Then we get the dominator tree as Figure 2d, where the parent node of every node is its immediate dominator.

Dominator trees are useful in various ways. As an example, a definition must dominate its uses, so a dominator tree can help verify legal use-define relations. And for optimizations, traversing a dominator tree provides a reasonable way to scan instructions in functions. And EarlyCSE pass performs a deep first search (DFS) on the dominator tree while finding and optimizing instructions.

3.1.3 LLVM EarlyCSE

Finally, we come to the actual CSE implementation in LLVM. As stated before, EarlyCSE aims to delete trivially redundant instructions. For our thesis work, it catches and optimizes all three patterns. We can use a simple example to indicate how it works:

```
1 int a, b;
2 int main(){
3     a = 1;
4     a = 2;  // OW
5     b = a;  // RAW
6 }
```

This non-atomic example contains two patterns that EarlyCSE may catch: OW (`a = 2;` overwrites `a = 1;`) and RAW (`b = a;` reads the constant value `2` written by `a = 2;`). Note that such descriptions are for understanding. To be more precise, optimizing passes work on IR files instead of source code. After being processed by the compiler front-end, the IR file will be:

```
1 ...
2 define i32 @main() {
3 entry:
4   store i32 1, i32* @a, align 4
5   store i32 2, i32* @a, align 4
6   %0 = load i32, i32* @a, align 4
7   store i32 %0, i32* @b, align 4
8   ret i32 0
9 }
```

This example has no loops or branches, containing only one basic block. So EarlyCSE just scans the instructions from the entry to the end. And in this process, EarlyCSE maintains two objects:

- `LastStore`. EarlyCSE uses a pointer, `LastStore`, to record the last seen store operation. When seeing a new store operation, it compares the new store with `LastStore`. If `LastStore` is not null and writes to the exact location as the current store, `LastStore` is overwritten and can thus be removed. By doing so, OW patterns are optimized.

- `AvailableLoads`. `AvailableLoads` is a hash table recording the current values of all the simple scalar expressions. When EarlyCSE walks down the dominator tree, it looks to see if the current store or load is in `AvailableLoads`. If yes, it updates the values. And if not, it inserts the read and written value. With a well-maintained `AvailableLoads`, we can replace load operations with values we have recorded if available so that RAR and RAW patterns can be optimized. One thing to note is that values in `AvailableLoads` are associated with a `Generation`. When walking down the dominator tree, a `CurrentGeneration` is also maintained, counting the version of the current

memory. By comparing `CurrentGeneration` and `Generation` associated with recorded values, EarlyCSE pass checks the validity of `AvailableLoads`. A more detailed algorithm description follows later.

These are the core helpers in EarlyCSE. And for our example, the process is shown in Figure 3:

- In the beginning, LastStore is null, and `CurrentGeneration` is 1 at first as Figure 3a.
- In Figure 3a, the first line `a = 1;` is a store operation, so `CurrentGeneration` is updated to 2. The value 1 of the variable `a` is recorded in `AvailableLoads` with the current generation 2. And `LastStore` is set to `a = 1;` as Figure 3b.
- In Figure 3b, current instruction `a = 2;` is a write, and writes to the same location as `LastStore`. So `LastStore`, `a = 1;`, is eliminated. Also, since the current instruction is a write, memory generation should be updated, and the new value of `a` is recorded in `AvailableLoads` as Figure 3c.
- In Figure 3c, The current instruction `r0 = a;` reads the value of `a`. According to `AvailableLoads`, `a` has an available value 2. All uses of the local variable `r0` can be replaced with the constant value 2. And this read `r0 = a;` becomes redundant and removed as Figure 3d.
- Eventually in Figure 3d, we have two instructions removed: an overwritten write and a read-after-write.

This is how EarlyCSE pass optimizes programs, we use this simple example to provide an intuition of the working process of it.

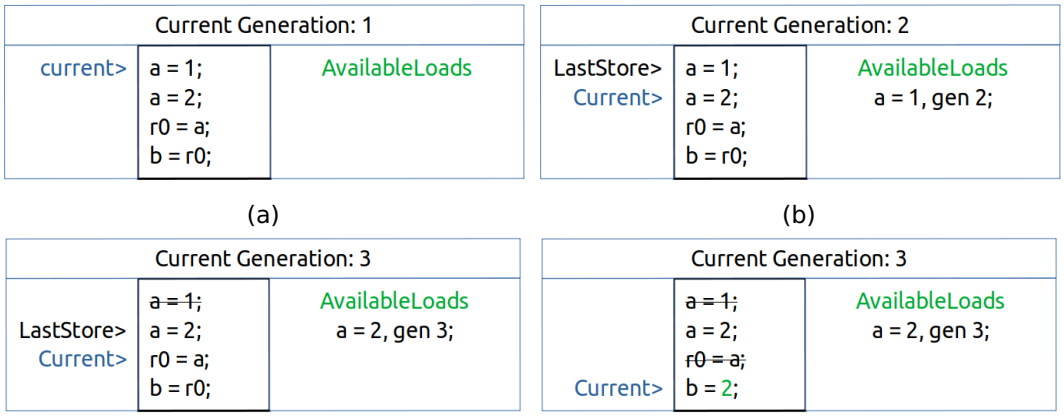


Figure 3: Process Optimizing a Non-atomic Example

3.2 Current Approach of EarlyCSE

In this section, we discuss EarlyCSE pass in more detail. We will describe its algorithms, and explain how it forbids optimizations of atomic accesses.

3.2.1 Algorithm Description EarlyCSE

Let's first take a look at the algorithms of EarlyCSE.

Algorithm 1: EarlyCSE

```
1  nodes ← [ ]
   // nodes is a stack used to DFS the dominator tree.
2  Construct FirstNode with AvailableLoads, CurrentGeneration, Root of
     DomTree
3  Push FirstNode onto nodes
4  while nodes is not empty do
5      Pop Node out of nodes
6      CurrentGeneration ← Generation of Node
7      if Node is not processed then
           /* Process the node.                                          */
8          processNode(Node)
9          childGeneration of Node ← CurrentGeneration
10     else if Not all children of Node are processed then
11         Push next child of Node onto nodes
12     else
           /* Node and all its children are processed, delete it.        */
13         Delete Node
```

The basic algorithm of EarlyCSE is a depth-first search (DFS) on the dominator tree. `CurrentGeneration` and `AvailableLoads` are transported across different tree nodes to enable some inter-nodes optimizations. And the key function of it is `processNode()`, where the real transformations take place. The actual procedure of `processNode()` is complicated, we extract the interesting parts as shown in Algorithm 2.

The helper functions used in Algorithm 2 are shown in Algorithm 3, Algorithm 4, and Algorithm 5:

- `mayReadFromMemory` and `mayWriteToMemory`. These two functions check whether the input instruction may read from or write to memory.
- `getMatchingValue`. This function is used here to check whether the found value in `AvailableLoads` can be used to replace the current instruction `Inst`. In fact, this is quite a simple function. It may look lengthy and confusing because it is also reused for "write back DSE", which checks whether a store writes precisely the same value back to a known location. We are not interested in write back DSE here, hence we omitted the according parts.
- `isSameGeneration`. This does just its name, checking whether the found loaded value is of the same memory generation with current instruction `Inst`.

Algorithm 2: EarlyCSE::processNode

```
   Input: Node
 1 BB ← BasicBlock of Node
 2 if BB has more than one predecessor then
 3 |  CurrentGeneration ← CurrentGeneration + 1
 4 if BB has exactly one predecessor && the predecessor ends with a
     conditional branch then
 5 |  Infer value based on the branch condition
 6 LastStore ← null
   /* Elimination Iteration                                          */
 7 for Inst in instruction list of BB do
 8 |  if Inst is trivially Dead then
 9 |  |  Remove Inst
10 |  if Inst is load then
11 |  |  if Inst is volatile or Inst is ordered then
12 |  |  |  LastStore ← null
13 |  |  |  CurrentGeneration ← CurrentGeneration + 1
14 |  |  InVal ← lookup pointer operand of Inst in AvailableLoads
15 |  |  Op ← getMatchingValue(InVal, Inst, CurrentGeneration)
16 |  |  if Op is not null then
17 |  |  |  Replace all uses of Inst with Op
18 |  |  |  Remove Inst
19 |  |  |  continue
20 |  |  Insert Inst together with CurrentGeneration into AvailableLoads
21 |  |  LastStore ← null
22 |  |  continue
23 |  if mayReadFromMemory(Inst) then
24 |  |  LastStore ← null
25 |  if mayWriteToMemory(Inst) then
26 |  |  CurrentGeneration ← CurrentGeneration + 1
27 |  |  if Inst is store then
28 |  |  |  if LastStore is not null and overridingStores(LastStore, Inst)
   |  |  |   then
29 |  |  |  |  Remove LastStore
30 |  |  |  Insert Inst together with CurrentGeneration into
   |  |  |   AvailableLoads
31 |  |  |  if Inst is unordered and Inst is not volatile then
32 |  |  |  |  LastStore ← Inst
33 |  |  |  else
34 |  |  |  |  LastStore ← null
```

Algorithm 3: mayReadFromMemory

```
Input: Inst
switch type of Inst do
    case load do
        return true
    case store do
        return isUnordered(Inst)
    otherwise do
        return false// return false by default
```

Algorithm 4: mayWriteToMemory

```
Input: Inst
switch type of Inst do
    case store do
        return true
    case load do
        return isUnordered(Inst)
    otherwise do
        return false// return false by default
```

Algorithm 5: EarlyCSE::getMatchingValue

```
Input: InVal, Inst, CurrentGeneration
if Inst is volatile or Inst is ordered then
    return null
if Inst is load then
    Matching ←DefInst of InVal
    Other ← Inst;
else
    Matching ← Inst
    Other ←DefInst of InVal
if Inst is store then
    Result ← Matching
else
    Result ← null
if Inst is store and DefInst of InVal != Result then
    return null
if !isSameMemGeneration(InVal, Inst) then
    return null
if Result is null then
    Result ← Matching
return Result
```

Algorithm 6: EarlyCSE::overridingStores

```
Input: Earlier, Later
if Earlier and Later write to different locations then
    return false
if Earlier is ordered or Later is ordered then
    return false
return true
```

Now we introduce Algorithm 2 in detail with a (non-atomic) example:

```
1  int a, b;
2  bool c, d;
3  ...
4  void func(){
5      a = 1;
6      if(d){
7          c = d;
8          a = 2;
9          a = 3;
10     } else {
11         b = a;
12         a = 4;
13     }
14     a = 5;
15 }
```

Listing 2: "A non-atomic example with control flow for EarlyCSE"

To put everything more clearly, we draw the function as a CFG, and show the optimizing process in Figure 4:

- At the beginning, as shown in Figure 4a, `CurrentGeneration` is initialized to 1.
- In Node 0, we just have a store to `a`. It hits the condition on Line 25 of Algorithm 2, so the memory generation is first updated, and this written value is inserted into `AvailableLoads`. Theoretically, since `a=1` is a write, `LastStore` should be set to it. But Node 0 has only one instruction(statement in fact), we will soon go to the next node. And please note Line 6, it means that `LastStore` is reset to null when processing a new node. So we omitted `LastStore` here. We then traverse child nodes of Node 0, starting from Node 1, which is the true branch of the condition `if(b)`. `CurrentGeneration` and `AvailableLoads` inherit from the parent node, Node 0. as shown in Figure 4b
- Now we are dealing with the instruction `c=b` in `EarlyCSE::processNode`, from line 2 to line 5. This is how `EarlyCSE` transports information between different basic blocks in a simple way. Here, Node 1 has exactly one predecessor, Node 0, and Node 0 ends with a conditional branch, `if(b)`. So based on these facts, we can infer that, given we have reached Node 1, the condition variable `b` must have the value `true`. So we can replace `b` here with `true`. And of course, `c=true` is a write, so update memory generation and record it in `AvailableValues` as shown in Figure 4c.
- Next, current instruction is `a=2`. This is a simple write, so we update the current generation, record this value, and also update `LastStore` to `a=2`, as shown in Figure 4d.
- And we come to the last instruction `a=3` in this branch. Again, it hits the condition on Line 25. And this time, `LastStore` is not null, and `overridingStores` obviously will return true here. So `LastStore`, which is `a=2` will be removed. And `a=3` is also a store itself, so memory generation will be updated, as well as `AvailableLoads`, as Figure 4e.
- We are done with the true branch, and come to the false branch. Again, we first inherit memory generation and `AvailableLoads` from the parent node. The current statement `b = a;` will be compiled into two instructions: a load from `a`, `r=a`, and a store which writes the loaded value to `b`, `b=r`. So first, `r=a` will hit the condition on Line 10. And after looking up in `AvailableLoads`, we can find that there exists a version of `a` with value 1. In `getMatchingValue` on Line 15, the memory generations are checked, and `a=1` is confirmed to be valid. So uses of `a` can be replaced with value 1. The load from `a` will also

be removed. The modified write `b = 1` will be processed. And the result is shown in Figure 4f.

- In Figure 4f, we deal with `a=4;`. It is just another simple write. Update generation, record value, and go on, as Figure 4g.
- Finally, we come to the last node, Node 3. Note that we used CFG in these figures for convenience. But EarlyCSE, as stated earlier, walks down the dominator trees instead of CFGs. And Node 3 is the third child node of Node 0 in the dominator tree. So the generation and `AvailableLoads` are inherited from Node 0 instead of Node 1 or 2. There is only one instruction in Node 3. So no more optimizations are possible, we just end our walk. And this will be the output of the EarlyCSE pass in this invocation.

3.2.2 Barriers of Atomics in EarlyCSE

With detailed algorithms of EarlyCSE in hand. We can briefly summarize how it works to optimize RAR, RAW, and OW:

- EalyCSE maintains `LastStore` to record the last seen store operation, so that when seeing a new store writing to the same location, it can remove the old one. In this way, EarlyCSE optimizes OW.
- EarlyCSE maintains `AvailableLoads` to record every known loaded or written value with the current generation. When trying to load a new value, it first checks whether there is a valid version of this value in `AvailableLoads`, and reuses the available value if possible. In this way, it optimizes RAR and RAW.
- To make the optimizations above correct, EarlyCSE needs to maintain the validity of written/read values. To do this, it maintains `CurrentGeneration` and `LastStore` carefully. Whenever there is possibly a write operation, it updates the generation number so that next time `isSameGeneration` will return false, meaning that we cannot use any value from `AvailableLoads` then. Even a store to an unrelated location will invalidate the whole memory. Similarly, `LastStore` is reset to null whenever there is possibly a read from memory, even from an unrelated location. Apparently, such checks are sound but not precise. EarlyCSE chooses such methods because it aims to do a fast walk "early" through the code so that trivial cases can be first found and optimized. And later passes, which we will introduce in the following sections, will take care of more complex cases.

And we marked the parts, which forbid algorithms from optimizing atomics, in red. Typically, these barriers are quite obvious and straightforward. Consider the following dummy program:

```
#include<atomic>
using namespace std;
atomic<int> a;
...

void func(){
a.store(1, memory_order_relaxed);
a.store(2, memory_order_relaxed);
}
```

Listing 3: A Dummy Example of Atomic OW

Such a simple OW pattern will not be optimized, because:

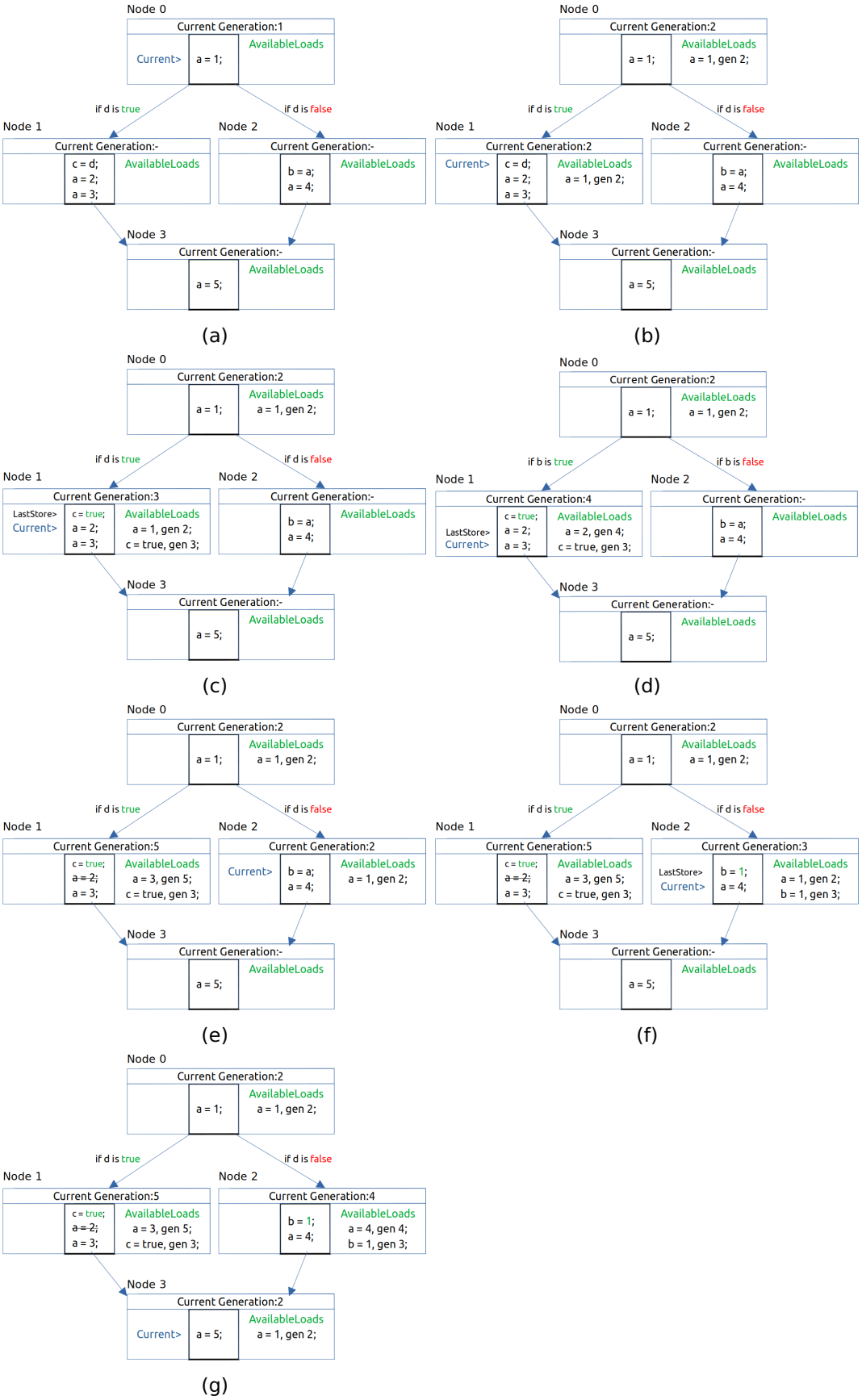


Figure 4: Process Optimizing a Non-atomic Example With Control Flow

- After traversing the store instruction on Line 6, we hope that `LastStore` will be set to it. So when walking down to the next write, we can remove this dead store. However, since such a store is atomic (ordered), it will hit the else branch in Algorithm 2 on Line 6. And `LastStore` will stay null. When reaching the second write on Line 7, there is no `LastStore` to be overwritten.
- What's more, even if `LastStore` is successfully set to `a.store(1, memory_order_relaxed)`, when processing `a.store(2, memory_order_relaxed)` in the next iteration, such an atomic write operation hits the condition in Algorithm 2 on Line 23, and `LastStore` is set to null "again" before used. LLVM is conservative and all atomic operations, including writes, will be taken as "may read from memory".
- Imagine our `LastStore` can magically survive through the barriers above and get used. In the next check, which resides in `overridingStores`, condition on Line 3 in Algorithm 6 will be hit and a false will be returned. As a result, our attempt to optimize such an overwritten write will fail.

From this simple enough example, we can find a number of checks in algorithms in EarlyCSE, ruling out atomic operations. We can see that the designer of this pass tried hard to catch every atomic and forbid them from being transformed. We did not mention all these checks because they resemble.

In order to enable optimizations on atomics, we need first to remove/modify these checks, so that atomics can be treated just as regular reads and writes. But as we mentioned earlier, these conservative checks are somehow reasonable. Additional constraints shall be added to make transformations safe.

3.3 Proposed Approach for EarlyCSE

Now, we present our approach for EarlyCSE. The basic structure of EarlyCSE is not modified. Algorithm 1 stays the same, and we modified parts of other Algorithms and list them in Algorithms 9, Algorithm 7, and Algorithm 8. We suffixed our methods with OA, which means "Optimize Atomic". Our modifications are marked in cyan.

To explain our methods, let's first start with the helper functions.

`getMatchingValueOA` is our version of `getMatchingValue`, shown in Algorithm 7. There are two modifications. First, on Line 1, we removed the checks which forbid atomics (but the check for volatile is reserved there). And on Line 15, we added new constraints there. This one may look a bit confusing, we can explain it here. The condition `Inst = Other` follows from the assignment on Line 5. If the condition is hit, then `Inst` must be a load operation, and `Matching` is a store or load which `Inst` reads from. According to Section 2.4, atomic RAW and RAR can only be optimized when the later read has a stronger or equal ordering with the earlier read/write. So if `Other` has a stronger ordering, we can only return null.

Algorithm 7: EarlyCSE::getMatchingValueOA

```
   Input: InVal, Inst, CurrentGeneration
 1 if Inst is volatile then
 2     return null
 3 if Inst is load then
 4     Matching ←DefInst of InVal
 5     Other ← Inst;
 6 else
 7     Matching ← Inst
 8     Other ←DefInst of InVal
 9 if Inst is store then
10     Result ← Matching
11 else
12     Result ← null
13 if Inst is store and DefInst of InVal != Result then
14     return null
15 if Inst = Other then
16     if Other has a stronger ordering than Matching then
17         return null
18 if !isSameMemGeneration(InVal, Inst) then
19     return null
20 if Result is null then
21     Result ← Matching
22 return Result
```

Algorithm 8: EarlyCSE::overridingStoresOA

```
   Input: Earlier, Later
 1 if Earlier and Later write to different locations then
 2     return false
 3 if Earlier has a stronger ordering than Later then
 4     return false
 5 return true
```

Similarly, for atomic OW patterns, we can only optimize when the later write has a stronger or equal ordering. So for `overridingStores`, we removed the checks preventing atomics and added constraints as cyan parts in Algorithm 8.

Algorithm 9: EarlyCSE::processNodeOA

```
   Input: Node
 1 BB ← BasicBlock of Node
 2 if BB has more than one predecessor then
 3 |  CurrentGeneration ← CurrentGeneration + 1
 4 if BB has exactly one predecessor && the predecessor ends with a
     conditional branch then
 5 |  Infer value based on the branch condition
 6 LastStore ← null
   /* Elimination Iteration                                          */
 7 for Inst in instruction list of BB do
 8 |  if Inst is trivially Dead then
 9 |  |  Remove Inst
10 |  if Inst is load then
11 |  |  if Inst is volatile or Inst is ordered then
12 |  |  |  LastStore ← null
13 |  |  |  if Inst is volatile then
14 |  |  |  |  CurrentGeneration ← CurrentGeneration + 1
15 |  |  InVal ← lookup pointer operand of Inst in AvailableLoads
16 |  |  Op ← getMatchingValueOA(InVal, Inst, CurrentGeneration)
17 |  |  if Op is not null then
18 |  |  |  Replace all uses of Inst with Op
19 |  |  |  Remove Inst
20 |  |  |  continue
21 |  |  if Inst is ordered then
22 |  |  |  CurrentGeneration ← CurrentGeneration + 1
23 |  |  Insert Inst together with CurrentGeneration into AvailableLoads
24 |  |  LastStore ← null
25 |  |  continue
26 |  if mayReadFromMemory(Inst) then
27 |  |  if Inst is not Store then
28 |  |  |  LastStore ← null
29 |  if mayWriteToMemory(Inst) then
30 |  |  CurrentGeneration ← CurrentGeneration + 1
31 |  |  if Inst is store then
32 |  |  |  if LastStore is not null and
   |  |  |    overridingStoresOA(LastStore, Inst)  then
33 |  |  |  |  Remove LastStore
34 |  |  |  Insert Inst together with CurrentGeneration into
   |  |  |    AvailableLoads
35 |  |  |  if Inst is volatile then
36 |  |  |  |  LastStore ← null
37 |  |  |  else if Inst is ordered then
38 |  |  |  |  LastStore ← null
39 |  |  |  |  if Inst is Store then
40 |  |  |  |  |  LastStore ← Inst
41 |  |  |  else
42 |  |  |  |  LastStore ← Inst
```

Then we can come to the major function `processNodeOA` shown in Algorithm 9.

The modifications include:

- We replaced the functions `getMatchingValue` and `overridingStores` with our version.
- For the problem that atomic stores are deemed as reads from memory, we added a condition in the branch on Line 26, so that atomic stores can escape. We do not choose to modify `mayReadFromMemory` directly, because, in practice, this function is implemented in `llvm/lib/IR/instruction.cpp`. This file is a core one and is widely used across the LLVM project. We would prefer to modify codes only used in the current file.
- A more complicated modification is from Line 35. This part aims to set `LastStore` given that the current instruction is a store. In the original LLVM implementation, if `Inst` is ordered or volatile, `LastStore` will be set to null so that `Inst` will not be removed even if it is overwritten. And to change this, we made a more detailed conditional branch to isolate one case where `Inst` is an ordered store. We set `LastStore` to `Inst` in this case. For other cases, we keep it as original.
- The most interesting modification is in the branch from Line 10. The original LLVM implementation updates `CurrentGeneration` immediately as it finds current instruction is a volatile or ordered load. This action will invalidate all values in `AvailableLoads`. By doing so, later attempts to optimize RAR and RAW patterns, which can potentially remove `Inst` will never succeed.

  To enable optimizations, we need to remove the ordered cases from the check, as we have done on Line 13. But simply doing so is not enough. Consider the following example:

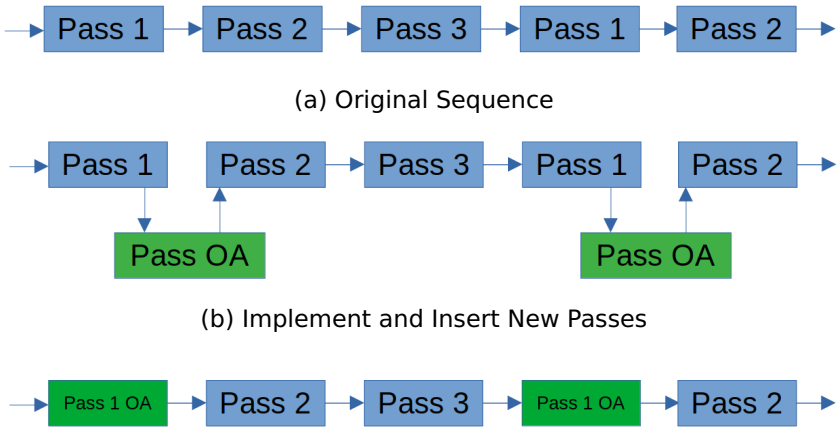
  To avoid such errors, we "deferred" the updating of `CurrentGeneration` for atomic cases to Line 21.

With all these modifications, we are done with EarlyCSE! And now, our EarlyCSE pass can optimize atomic patterns safely.

4 From EarlyCSE to Others

Before continuing to introduce our methods for other passes, we would like to summarize the work on EarlyCSE first here. We organize the previous content so that it can be understood more easily. The following content is similar to EarlyCSE in principle, and this summary can also help understand the subsequent sections.

4.1 Design Choice of Our Implementation



(a) Original Sequence

(b) Implement and Insert New Passes

(c) Modify Passes in Place

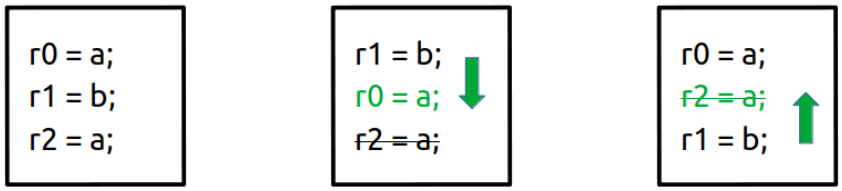
Figure 5: Choices of Implementation

If the original sequence of passes in one go of compilation is as Figure 5a, basically, we have two choices to enable optimizations for atomics. The first choice is to implement a totally new pass `Pass OA`, and invoke it in the sequence of compilation. The main advantage of this method is that we do not need to know about or change any existing passes. We only need to implement a dedicated pass for atomic operations. However, the optimizations of RAR, RAW, and OW are nothing new for the compiler, though we are introducing new guests for them. So implementing a new pass can still duplicate lots of existing codes from old ones. What's more, one pass is not only invoked once in a compiling sequence. A fully optimizing pass for atomics, invoked several times as shown in Figure 5b can significantly increase the compiling time.

So our choice is the second method, modifying existing passes as Figure 5c. This requires us to look at a lot of codes. But we can preserve the existing complexity of algorithms, and reuse current codes much as possible. This is also an important principle during our implementation: minimal changes. LLVM is really large a code base. There are many unknown areas to us, even if we already had a tour around. So in order not to raise unknown errors, and not hurt performance significantly, we choose to make modifications under control. The `mayReadFromMemory` used in EarlyCSE is an instance of such complex dependencies (However, `mayReadFromMemory` does need to be refined, which is another story).

4.2 Not All Possible Transformations are Enabled

Choosing to modify existing passes will also miss a part of the chances of optimizations. Consider the following example: Here is a simple non-atomic RAR

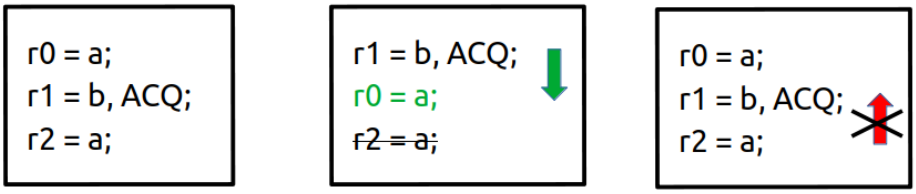


(a) Original (b) Reordering 1 (c) Reordering 2

Figure 6: Reordering Choices of Non-atomic RAR

case shown in Figure 6a. The second read from `a` is obviously redundant and can be removed. To achieve this, we have two choices as Figure 6b and Figure 6c. We can move the first read, `r0 = a`, down, or the second read, `r2 = a`, up. Two ways will both remove the redundant `r2 = a` successfully. Just the orders of the remaining instructions are different.

However, reordering non-atomic cases is not free. Let's replace the read from `b` with an atomic read with acquire ordering as Figure 7. Since we can only move instructions from before to after an acquire read, we have only one choice here as Figure 7b.



(a) Original (b) Reordering (c) Not Allowed

Figure 7: Reordering Choices of Atomic RAR

The problem is that the current LLVM tries to reserve the original relative order of instructions when performing eliminations. For instance, it chooses Figure 6c for the given example, so that the remaining instructions keep the original relative order. However, in an atomic case, such reordering is not allowed as Figure 7c. So if we follow the current algorithm (and we did), nothing will happen to the program in Figure 7a, even though it is possible to be optimized.

4.3 Types of Modifications Made

With EarlyCSE as an example, one can find that our modifications are of two types:

- First, help atomics to escape. The example shown in Section 3.2.2 gives an image of how many barriers there are forbidding atomics. Typically, these are explicit conditions ruling out ordered operations. But sometimes, these checks hide deeply in functions from other libraries. We need to refine the conditional branches, or re-implement some functions if necessary, to allow atomics to be served as regular memory accesses.

- And more importantly, we need to add new constraints to make transformations safe. Directly enforcing passes to transform atomic patterns definitely brings problems. After helping atomics out, we need to adapt the algorithms of passes to put confinements on atomics according to rules in section 2.4. This is also one main challenge of our project.

4.4 Challenges of This Work

The challenges mainly come when add constraints for atomics. And the main problem is that most algorithms of the current LLVM are designed for non-atomic cases. In fact, Section 4.2 has shown a part of this problem. The scanning order of non-atomic accesses does not matter much, because non-atomic accesses can be reordered relatively freely. So when designing the algorithms, one may just choose a preferred order. However, the ordering constraints of atomic accesses do not allow free reordering. We need to adapt these constraints to the existing algorithms.

On the other hand, because of the current conservative attitude towards atomics, when a pass encounters an atomic access, it has only two choices: skip it, or just stop scanning. However, if we choose to optimize the atomics, we have to do something more. We need to check, for example, the ordering of this instruction, and the location of the memory access. And we may need to eliminate this atomic access, but not just skip or stop.

4.5 Some More Explanation

One may find in the example in Figure 4 that even in the end, the code is not fully optimized. `a = 5;` in Node 3 totally dominates `a = 1;` in Node 0. So `a = 1;` is overwritten and shall be removed. But it was not. The direct reason for this is that `LastStore` is only valid inside a single node. So a write in another node may never kill a write in the current one. And essentially, this is because EarlyCSE, as stated before, is designed to be fast. It only performs trivial optimizations. So EarlyCSE goes through instructions only once and never looks back. It will not modify Node 0 after leaving Node 0. And this is why works in the following sections are required. They take care of the (more complex) cases which EarlyCSE cannot handle.

Last but not least, passes in LLVM depend on a number of libraries. Even if we are only trying to introduce a small fraction of LLVM, a lot of dependent libraries or structures get involved. If some parts of the shown algorithms look a bit weird, this might be the reason. And as we mentioned, passes in such a product-level compiler are supposed to cover all corner cases and have numerous, not interesting details which we omitted. We attempt to explain our technical details much as we could. Nevertheless, covering everything is not feasible, and not necessary. In order to explain more concisely and understandably, we may have omitted uninteresting details, changed variable names, simplified procedures, inlined functions, etc. But the essential algorithms and procedures remain. The principles of our modifications are pretty simple, and our descriptions are to show these basic ideas of the proposed methods. And We wish these would be enough to help understand our concepts.

In the following two sections (Section 5 and Section 6), we go on to introduce our solutions to the current LLVM.

5 InstCombine

In this section, we introduce our works with LLVM InstCombine Pass. To be more specific, RAR and RAW patterns are taken care of in this pass. Unlike EarlyCSE, InstCombine Pass makes use of alias analysis when optimizing these patterns, which enables it to perform optimizations on more complicated memory accesses.

5.1 Overview InstCombine

5.1.1 Instruction Combining

Instruction combining is a type of optimization that combine instructions to form fewer and simpler instructions. As a dummy example, the following two lines can be combined into one.

```
a += 1;
a += 1;
```

After instruction combining, we get:

```
a += 2;
```

Apparently, instruction combining may serve a great number of instructions, from numerical operations like addition and multiplication to logical operations. And typically these optimizations are trivial and straightforward.

5.1.2 LLVM InstCombine

LLVM InstCombine is the LLVM implementation of such optimizations. In fact, LLVM InstCombine is a large, diverse collection of "peephole optimizations" that (typically) transforms instructions into more efficient forms. The optimizations performed in LLVM InstCombine cover various types and go much beyond the example shown above. Our target patterns, RAR and RAW, are included in them. So we shall find the (small) fraction concerning our interesting problems, and adapt it. Again, let's start with a non-atomic example:

```
int x, y;
int a, b, c;
void func(){
    x = 1;
    y = 2;
    a = x;
    b = y;
    c = x;
}
```

LLVM InstCombine uses a "worklist-driven" algorithm to run the optimizations. This kind of algorithm is widely used among various passes in LLVM. Typically, a pass first forms a worklist from a function or a basic block. Then it takes an instruction out of the worklist, optimizes it, and inserts the optimized instruction(s) (or nothing, if the instruction ends up eliminated) back into the worklist. Such iterations repeat until some condition is fulfilled. In the given example, the code is compiled into the instructions in Figure 8a. As we mentioned in Section 3.2.1, a "copy" statement like `a = x;` will be transformed into two instructions, `r0 = x;` and `a = r0;` in an IR file. The procedure of InstCobine pass optimizing such a function could be:

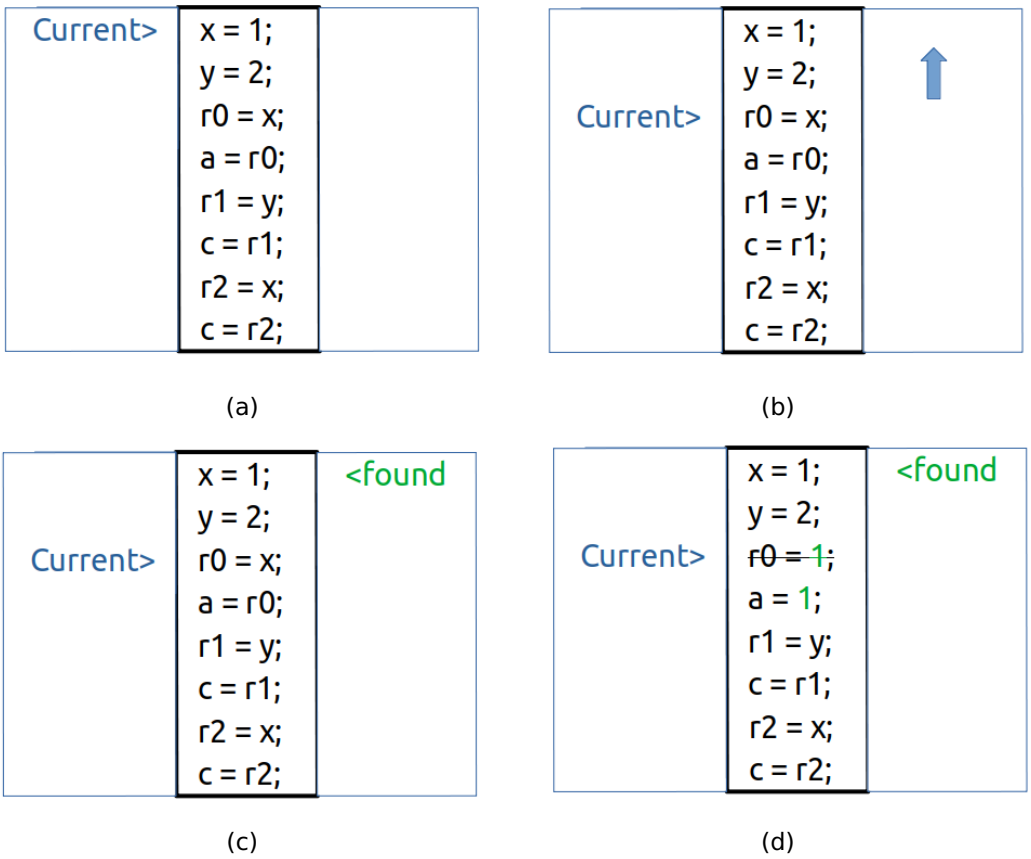


Figure 8: Process of InstCombine Pass Optimizing a Non-atomic Example

- To start with, InstCombine makes a worklist out of the function. In this example, all the instructions will be inserted into the worklist. InstCombine pops instructions out of the worklist in order. The initial state is Figure 8a.
- For each instruction, it invokes a "visit" function to check if the current instruction could be transformed or eliminated. For our example, since we are working on memory accesses, or more specifically, RAR and RAW here, we first show a "visit" to `r0 = x;` as in Figure 8b.
- When visiting a load instruction, InstCombine searches backward as Figure 8b, to find if there is an available load so that the current one can be replaced or deleted. For `r0 = x;`, InstCombine will start from `y = 2;`. This is a store, which means it could modify memory. So InstCombine will check if `y = 2;` may modify the memory location(s) read by `r0 = x;`. After finding there is no overlap in memory, it will continue searching. And finally, it will find a write `x = 1;`, which writes to the location where `r0 = x;` reads from, as in Figure 8c.
- After finding the available value, InstCombine will replace all uses of `r0` with the found value `1`. And if `r0` is then never used, this instruction can be then removed, as Figure 8d.
- Then InstCombine will continue processing instructions in the worklist until all of them are visited. The process resembles the treatment of `r0 = x;`, and we skip them for now.

Here one may find the difference between EarlyCSE and InstCombine. When trying to optimize a RAR or RAW pattern, EarlyCSE will give up whenever there

is a write (like `y = 2;` in this example) to memory, while InstCombine will check if the write really modifies the interesting location. This is why InstCombine can deal with more complex patterns. And this is also one of the reasons that EarlyCSE is necessary: it does not need careful comparison and checks of memory accesses, so it can quickly optimize trivial cases where these checks are not necessary.

5.1.3 LLVM Alias Analysis

Such memory access checking performed here is called alias analysis [5]. Alias analysis is an important technique in compilers. Pointers are essentially aliases of memory locations. And two pointers are said to be aliased if they point to the same location. Alias analysis attempts to determine whether or not two pointers ever can point to the same object in memory. The class of alias analysis has covered a big range of algorithms, and there are various ways to classify them: flow-sensitive vs. flow-insensitive, context-sensitive vs. context-insensitive, field-sensitive vs. field-insensitive, etc. And typically, alias analyses will tell if two pointers must, may, or no alias. Respectively, this means the two pointers always point to the identical object, might point to the same object, or cannot point to the same object.

The LLVM Alias Analysis is mostly a flow-sensitive one. Just as the tradition, it responds must, may, or no. And in addition, it can tell mod/ref information, which says whether an operation (must, may, or never) reads or writes from a location. LLVM Alias Analysis is the basis of many powerful and useful optimizations performed in LLVM. And obviously here in InstCombine, we make use of it, since it helps check the memory accesses in the way of searching (like `y = 2;`) could be a clobber (if they modify the value of `x`).

5.2 Current Approach of InstCombine

Now we provide the detailed algorithms of InstCombine, see how they work, and why they forbid atomics.

5.2.1 Algorithm Description InstCombine

First of all, the basic algorithm of InstCombine, as stated, is a worklist-driven algorithm as Algorithm 10, and helper functions used by it are shown in respective algorithms.

One thing interesting here is the `visit` function. Typically, functions like `visit` may have all kinds of instructions in LLVM as the input parameter. And there are too many kinds of instructions in LLVM. To avoid writing a much too verbose switch-case statement, LLVM implements an "Instruction visitor" using the C++ macro. The details of such implementation are not relevant here and out of our scope of discussion. And the result is, `visit` will invoke the corresponding visiting function for the input instruction. For our RAR and RAW patterns, we are always visiting the load instructions, thus we need to look into the `visitLoadInst` function as shown in Algorithm 13.

Algorithm 10: InstCombine

```
   Input: Function
1  Worklist ← prepareICWorklistFromFunction(Function)
2  while Worklist is not empty do
3      Inst ← pop an instruction from Worklist
4      if isInstructionTriviallyDead(Inst) then
5          Remove Inst
6          continue
7      Result ← visit(Inst)
8      if Result is not null then
9          if Result != Inst then
10             Replace all uses of Inst with Result
11             Push users of Result to Worklist
12             Push Result to Worklist
13         else
14             if isInstructionTriviallyDead(Inst) then
15                 Remove Inst
16             else
17                 Push users of Inst to Worklist
18                 Push Inst to Worklist
```

Algorithm 11: isInstructionTriviallyDead

```
   Input: Inst
1  if Inst is used then
2      return false
3  if Inst is terminator then
4      return false
5  if mayHaveSideEffects(Inst) then
6      return false
7  return true
```

Algorithm 12: mayHaveSideEffects

```
   Input: Inst
1  if mayWriteToMemory(Inst)or Inst may throw then
2      return true
3  return false
```

Algorithm 13: visitLoadInst

```
   Input: LoadInst
1  Value ← FindAvailableLoadedValue(LoadInst)
2  if Value is not null then
3      Replace all uses of LoadInst with Value
4      return LoadInst
5  return null
```

Algorithm 14: FindAvailableLoadedValue

```
   Input: LoadInst
1  StrippedPtr ← Strip pointer of LoadInst
2  ScanBB ← Basic block of LoadInst
3  AtLeastAtomic ← Atomicity of LoadInst
4  if LoadInst is Ordered then
5      return null
6  Available ← null
7  MustNotAlias ← []
8  for Inst in range(instruction before LoadInst, start of ScanBB) do
9      Available ←
         getAvailableLoadStore(Inst, StrippedPtr, AtLeastAtomic)
10     if Available is not null then
11         break
12     if mayWriteToMemroy(Inst) then
13         Push Inst to MustNotAlias
14 if Available is not null then
15     Loc ← memory location of LoadInst
16     for Inst in MustNotAlias do
17         if mayModify(Inst, Loc) then
18             return null
19 return Available
```

Algorithm 15: getAvailableLoadStore

```
   Input: Inst, Ptr, AtLeastAtomic
1  if Inst is a Load then
2      if Inst has a weaker atomicity than AtLeastAtomic then
3          return null
4      LoadPtr ← Strip pointer of Inst
5      if LoadPtr and Ptr are the same addresses then
6          return Inst
7  if Inst is a Store then
8      if Inst has a weaker atomicity than AtLeastAtomic then
9          return null
10     StorePtr ← Strip pointer of Inst
11     if StorePtr and Ptr are different addresses then
12         return null
13     Value ← Value operand of Inst
14     return Value
15 return null
```

And lastly, There is a deep calling stack for alias analysis in LLVM, involving a series of methods in LLVM Alias Analysis. And listing all functions in the calling stack does not make much sense. For readability, we simplified these functions and combined them into the function shown as Algorithm 16. It determines whether an operation may modify the given memory location.

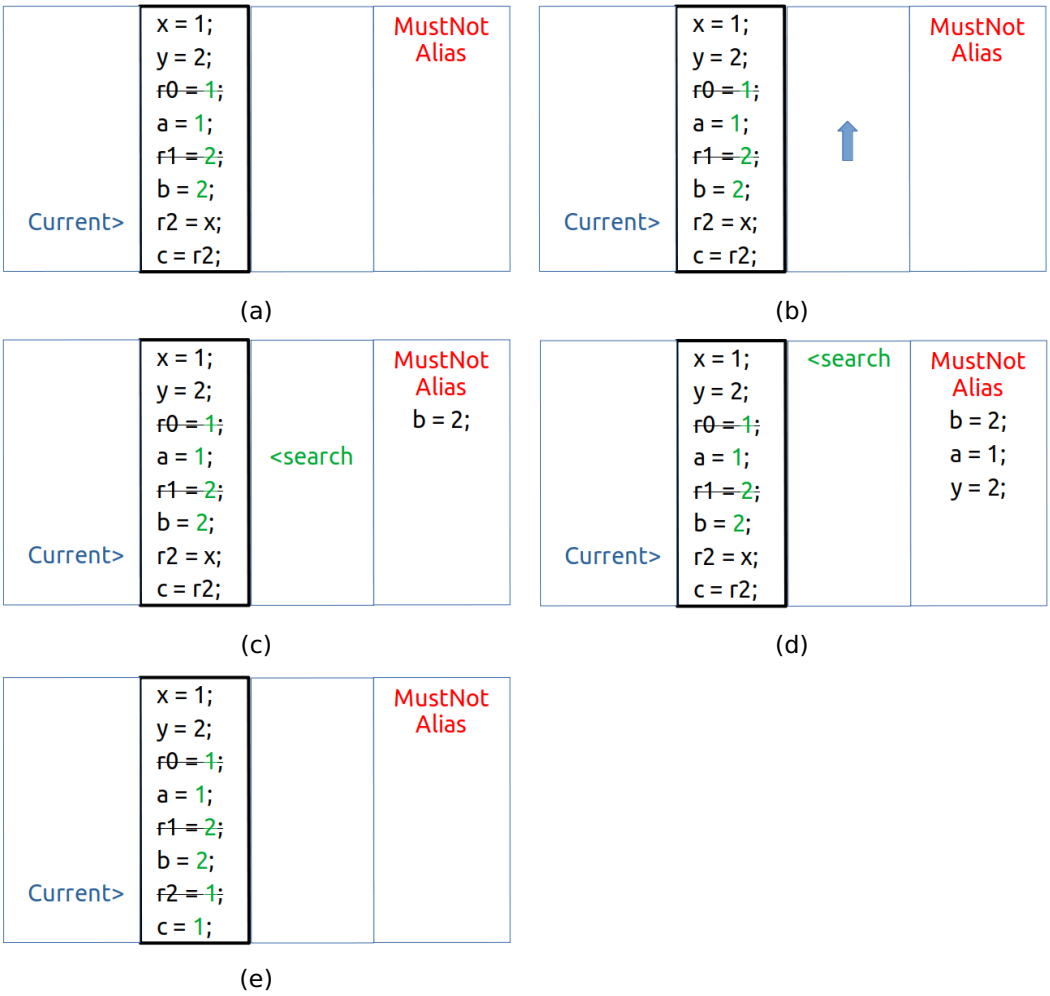


Figure 9: Detailed InstCombine Algorithms Optimizing a Non-atomic Example

Algorithm 16: mayModify

Input: *Inst*, *Loc*

```
1  if Inst is ordered then
2      return true
3  if Inst is load then
4      return false
5  if Loc is not null and Inst is store then
6      AR ← Alias Analysis Result of Inst at Loc
7      if AR is no alias then
8          return false
9      if AR is must alias then
10         return true
11 return true
```

Now we have all the used functions in hand, let's get back to the example in Figure 8 to see how these algorithms are running:

- To start with, we begin with InstCombine shown in Algorithm 10. We first create the worklist as Line 1.

  We did not show the details of `prepareICWorklistFromFunction`, since it is

verbose and not interesting. The main task of this function is to check the reachable instructions. For example, if there is a branch that will never be reached, instructions in that branch will not be inserted into the worklist. And as we mentioned, in our example, all instructions will be added.

- After forming a worklist, we can start the optimizing iteration, the while-loop. This iteration will repeat until the worklist is empty. In other words, the optimization will continue until all instructions are visited. A worklist, instead of direct iterating over the instructions, is used here, because optimizing and modifying an instruction may create new chances for optimizations in return. So we shall insert the modified or newly generated instructions back into the worklist, if any.

- For each iteration, we pop an instruction out of the worklist. Remove it if it is trivially dead. And visit it as Line 7. As we mentioned, visiting an instruction is polymorphic. The corresponding function will be automatically invoked based on the type of input instructions. However, the basic rule is that `visit` will return a pointer to an instruction.

  If the original instruction is not modified, then `visit` will return null, which means there is nothing to do, just continue to visit the next instruction.

  If the original instruction needs to be replaced with a new instruction, then the pointer to the new instruction will be returned. `Result` will point to the new one, which hits condition on Line 9. All uses of `Inst` will be replaced with `Result`, and `Result` together with its uses will be inserted back into the worklist.

  And if the original instruction is just modified, then the original pointer is returned, which hits the else branch on Line 13. This case resembles the last case. The difference is that now the modified instruction can be dead since all its uses are replaced. So check if we can remove it.

- Now we have explained the iteration structure. And we can continue to see how exactly an instruction is visited. Since every iteration is largely similar, we choose to skip some steps and come to the visit to `r2 = x;`. By far, the previous two reads have been optimized and removed. Writes to `a` and `c` also have been replaced with constant values as Figure 9a.

- `r2 = x;` is a load instruction, so `visitLoadInst` in Algorithm 13 is invoked. And according to it, we invoke `FindAvailableLoadedValue` to find if we have any available values for this load. And if there is, we can replace all uses of the load with the found value, and return the instruction back.

- In `FindAvailableLoadedValue` as shown in Algorithm 14, the input `LoadInst` is our current load instruction `r2 = x;`. first, we strip off pointer casts of the pointer operand of the load. In our example, the pointer `x` is quite simple and is just returned. Then, we get the current basic block `ScanBB` as our working scope. This also means InstCombine optimizes load operations inside basic blocks. There are no inter-basic-block behaviors. This is also why we used a straight-line program as our example. After that, we get `AtLeastAtomic` from `LoadInst`. It is a boolean value, just indicating whether `LoadInst` is atomic. And here it is false. `Available` is the to-be-found available loaded value and is now initialized to null. And `MustNotAlias` is a vector of instructions used to check whether instructions will clobber the optimization.

  Then we begin the for-loop. The iteration starts from the instruction right before `LoadInst`, `b = 2;`, and scans reversely to find an available value, as Figure 9b. We use a pointer, `search`, to indicate the instruction being checked by `FindAvailableLoadedValue`.

- Then we need to invoke `getAvailableLoadStore` as Algorithm 15. Basically, we can see this function has two conditional branches: one for loads, and one for stores. This exactly corresponds to our patterns to be optimized: load for RAR, and store for RAW. Now the input is `b = 2;`, a store to an unrelated location. So it finally hits the condition on Line 11. And the function will return null, which means the current instruction, `b = 2;`, does not provide an available value for `r2 = x;`.

- Now we are back to `FindAvailableLoadedValue`. Since `getAvailableLoadStore` returns null, we come to Line 12, and push `b = 2;` into `MustNotAlias`. Then continue the iteration, as Figure 9c.

- We are done with one invocation to `getAvailableLoadStore`. The rest of `FindAvailableLoadedValue` is similar. We will search all the way till we find `x = 1;`. And of course, if we cannot find an available value for `r2 = x;`, `Available` will stay null and be returned. As Figure 9d, we have found the available value at `x = 1;`. So now `Available` is not null, the condition on Line 14 is hit. We get the load location of `r2 = x;`, which is `x`. And compare it with every write in `MustNotAlias`.

  Now you can see the meaning of `MustNotAlias`, it records every write we have seen so far. And if `x` is written by any instruction in `MustNotAlias`, it means the value of `x` might have been modified, and cannot be used by `r2 = x;`. This is why this vector is named `MustNotAlias`: the instructions we checked before finding the available value cannot be aliased with it.

- At last, `vistLoadInst` will replace all uses of `r2 = x;` with the found write `x = 1;`, and the pointer to `LoadInst` is returned to `InstCombine`. And in `InstCombine`, because a modified instruction is returned, we come to the else branch on Line 13. `r2 = 1;` is now not used anywhere, and can be removed. We eventually get the result in Figure 9e.

This is a full run of Instcombine for RAR and RAW patterns. We covered most details and omitted some verbose or uninteresting ones. Though there are a lot of functions called, the principles of InstCombine are quite direct. In the following parts, we point out the barriers of atomics in these algorithms and propose our solutions.

5.2.2 Barriers of Atomics in InstCombine

In the process of introducing these algorithms, one may have already seen those barriers, now we just list them out. The barriers are marked red just as we did for EarlyCSE. And we can just pick out these barriers using a very simple atomic example in Figure 10.

- Currently, we are visiting an atomic acquire read, `r = x, ACQ;` as Figure 10a. We get into the calling stack, and our first barrier comes in Algorithm 14. On Line 4, since the current load is ordered, the whole function just returns null and terminated.

- In the iteration in Algorithm 14, we invoke `getAvailableLoadStore`. In fact, this function does not forbid atomics itself. Because the check in `FindAvailableLoadedValue` has made sure that no atomic can survive here. `AtLeastAtomic` is used to guarantee that We can value forward from an atomic to a non-atomic, but not the other way around. This is a conservative and correct treatment.

- Assume we returned null from `getAvailableLoadStore`. We are now on Line 12. A write like the current one, `y = 2, REL;`, will of course be inserted into

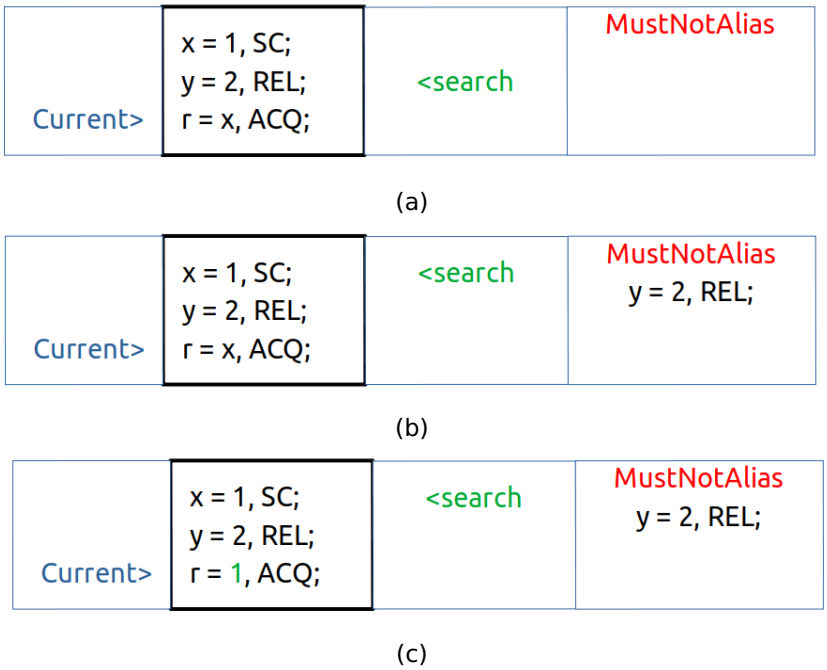


Figure 10: A Small Trip in InstCombine to Find Barriers

`MustNotAlias` as Figure 10b. However, imagine we have an atomic read, which does not write anything. It will still be inserted. This may sound similar. We dealt with this problem for EarlyCSE. And now the problem comes again: `mayWriteToMemory` takes all atomic operations as writes.

- After reaching `x = 1, SC;`, we need to check if its value is written by operations in `MustNotAlias`. This is the next barrier. In `mayModify`, on Line 1, an ordered operation is assumed to modify any memory location. So any atomic operation in `MustAlias` will stop the optimization, no matter if it is a read or a write, and no matter which memory location it is accessing.
- And assume this lucky invocation can successfully return to `visitLoadInst`. We will be able to replace uses of this load as Figure 10c. However, after returning to `InstCombine`, when we are trying to remove the unused instruction `r = 1, ACQ;`, we would fail. Since `isInstructionTriviallyDead` take atomic operations as having side effects. This, again, thanks to `mayWriteToMemory` as Algorithm 12. So even if the read `r = x, ACQ;` has been modified and unused, it will not be removed since it is not "trivially dead".

5.3 Proposed Approach for InstComibine

Now we got the barriers. As we talked about in Section 4, we will remove barriers and introduce new constraints. Again, we marked our modified parts as cyan. The new algorithms are suffixed with "OA", shown from Algorithm 17 to Algorithm 21. We introduce our modifications just in this order:

Algorithm 17: InstCombineOA

```
   Input: Function
1  Worklist ← prepareICWorklistFromFunction(Function)
2  while Worklist is not empty do
3      Inst ← pop an instruction from Worklist
4      if isInstructionTriviallyDead(Inst) then
5          Remove Inst
6          continue
7      Result ← visit(Inst)
8      if Result is not null then
9          if Result != Inst then
10             Replace all uses of Inst with Result
11             Push users of Result to Worklist
12             Push Result to Worklist
13         else
14             if isInstructionTriviallyDead(Inst) then
15                 Remove Inst
16             else if Inst is load and not used then
17                 Remove Inst
18             else
19                 Push users of Inst to Worklist
20                 Push Inst to Worklist
```

Algorithm 18: visitLoadInstOA

```
   Input: LoadInst
1  Value ← FindAvailableLoadedValueOA(LoadInst)
2  if Value is not null then
3      Replace all uses of LoadInst with Value
4      return LoadInst
5  return null
```

- For `InstCombineOA`, we added a small branch to allow removing the atomic instruction. This is like our treatment of EarlyCSE.

  core functions like `isInstructionTriviallyDead` are widely used across the LLVM project, and we cannot easily modify these functions. So we choose to use small conditions to help atomics escape.

- For `visitLoadInstOA`, we just replaced `FindAvailableLoadedValue` with a new version.

- For `FindAvailableLoadedValueOA`, we did several changes. First, we replaced `AtLeastAtomic` with `AtLeastOrdering`. Because we are now optimizing atomics, atomicity is not enough to provide ordering information.

  `getAvailableLoadStore` is replaced accordingly. The interesting part is from Line 10. We added two ordering constraints for checking ordering. Note the place where we choose to add them: we check the ordering after finding `Inst` does not provide an available value. If `Inst` is the value we are looking for, then `Inst` can form a RAR or RAW pattern with `LoadInst`. And if not, we must check if `Inst` is a clobber according to Table 1. And we break the iteration if it is. And we replaced `mayModify` with our version.

Algorithm 19: FindAvailableLoadedValueOA

```
   Input: LoadInst
 1 StrippedPtr ← Strip pointer of LoadInst
 2 ScanBB ← Basic block of LoadInst
 3 AtLeastOrdering ← Ordering of LoadInst
 4 Available ← null
 5 MustNotAlias ← []
 6 for Inst in range(instruction before LoadInst, start of ScanBB) do
 7     Available ←
         getAvailableLoadStoreOA(Inst, StrippedPtr, AtLeastOrdering)
 8     if Available is not null then
 9         break
10     if Inst is Load and has an ordering Stronger than or equal to acquire
        then
11         break
12     if Inst is Store then
13         if Both Inst and LoadInst are sequentially consistent then
14             break
15     if mayWriteToMemroy(Inst) then
16         Push Inst to MustNotAlias
17 if Available is not null then
18     Loc ← memory location of LoadInst
19     for Inst in MustNotAlias do
20         if mayModifyOA(Inst, Loc) then
21             return null
22 return Available
```

Algorithm 20: getAvailableLoadStoreOA

```
   Input: Inst, Ptr, AtLeastOrdering
 1 if Inst is a Load then
 2     if Inst has a weaker ordering than AtLeastOrdering then
 3         return null
 4     LoadPtr ← Strip pointer of Inst
 5     if LoadPtr and Ptr are the same addresses then
 6         return Inst
 7 if Inst is a Store then
 8     if Inst has a weaker atomicity than AtLeastOrdering then
 9         return null
10     StorePtr ← Strip pointer of Inst
11     if StorePtr and Ptr are different addresses then
12         return null
13     Value ← Value operand of Inst
14     return Value
15 return null
```

Algorithm 21: mayModifyOA

```
   Input: Inst, Loc
1  if Inst is load then
2      return false
3  if Loc is not null and Inst is store then
4      AR ← Alias Analysis Result of Inst at Loc
5      if AR is no alias then
6          return false
7      if AR is must alias then
8          return true
9  return true
```

- For `getAvailableLoadStore`, we changed the input parameter as we mentioned earlier. Instead of checking atomicity, we now check the relationship between orderings of `Inst` and `AtLeastOrdering` (essentially the ordering of *LoadInst*).
- And finally for `mayModify`, we simply removed the check of atomics. Please note that `mayModify` is a function we made up for explanation. The practical work we did is to create a series of (boring) new functions in LLVM Alias Analysis dedicated to our modifications. These new methods combined to be equivalent to removing an atomic check. So we represent our modifications as this.

These are our methods for `InstCombine`. We are now done with another important pass. And the work on `InstCombine` follows our summaries in Section 4:

- To enable LLVM passes to optimize atomics, we need to first find out and remove barriers. And then add new constraints to make optimizations sound.
- Existing algorithms are designed for non-atomics. So ordering of instructions is typically not considered. When implementing optimizations for atomics, we need to adapt algorithms to do these checks. However, such adaptions are quite feasible thanks to the good design of LLVM.
- The concept of avoiding atomics is deeply rooted in LLVM. IR functions like `mayWriteToMemory`, utilized functions like `isInstructionTriviallyDead`, and functions in LLVM Alias Analysis, all have been ensured to avoid atomics. These are additional obstacles in the way of optimizing atomics.

We can now continue to the last pass: DSE.

6 DSE Pass

DSE pass takes care of more complex OW patterns than EarlyCSE. Like InstCombine, it makes use of alias analysis to perform more precise optimizations. In addition, it uses MemorySSA to efficiently scan instructions. In this section, we introduce algorithms in the current DSE pass and our optimizations implemented.

6.1 Overview DSE

6.1.1 Dead Store Elimination

DSE stands for Dead Store Elimination (DSE). This is quite a straightforward optimization widely applied in compilers. As the name indicates, it intends to remove an assignment of a variable if the assigned value is not used by any subsequent instruction. A very simple example could be:

```
1 a = 1;
2 r = b;
3 a = 3;
4 print(a);
```

The write `a = 1;` on Line 1 is never used and could be removed. Typically, DSE optimizes the OW patterns. Like the dummy example above, `a = 1;` is overwritten by `a = 3;`, and there is no use of `a` between them. So we can determine that `a = 1;` is killed by `a = 3;`.

In fact, we have seen EarlyCSE optimizing OW patterns. But EarlyCSE is much too simple. For this example, the read `r = b;` will set `LastStore` back to null, so `a = 3;` cannot find the previous store to be killed. So it is obvious that we need alias analysis like we did in InstCombine, to precisely judge whether instructions between the two writes could be a clobber.

This is approximately the basic algorithm of DSE: for each store operation, we look upwards to find if it kills any previous store. And apparently here comes another problem: walking through the whole instruction list over and over again is inefficient and unnecessary. Overwritten Write (OWis between stores. It is more efficient to only scan store operations. DSE makes use of another tool in LLVM to do this: MemorySSA.

6.1.2 LLVM MemorySSA

MemorySSA is an analysis that allows us to cheaply reason about interactions between various memory accesses [13]. Intuitively, the version of the memory is updated when there is a store to the memory, while a load typically just uses the memory, but does not modify it. So a store defines a new version of the memory. On the other hand, memory versions are associated with their defining writes. And MemorySSA is used to maintain all these memory versions with their defining operations.

More generally, these operations defining the memory are called `MemoryDef`, including store, function calls, and those operations that may introduce ordering constraints, like memory fences and acquire (or higher) loads. Instructions that read from the memory, like regular loads, are called `MemoryUse`. And both `MemoryDef` and `MemoryUse` are `MemoryAccess`.

For example in the following example, we marked the memory definitions with their relations:

```
a = 1;       // 1 = MemoryDef(liveOnEntry)
r0 = a;
r1 = b;
b = 2;       // 2 = MemoryDef(1)
c = 3;       // 3 = MemoryDef(2)
```

Note that `a = MemoryDef(b)` means that MemoryDef `a`, mapped to the current instruction, defines a new memory version based on another MemoryDef `b`. In the example above, `b = 2;` is a store operation. It is numbered `2`, and it defines a new memory version based on the one defined by `1`, `a = 1;`. And as the first store operation, `a = 1;` defines a memory version based on a special `liveOnEntry`, which means `a = 1;` is the first store in the current scope.

So you can see that `MemoryDef` gives a single chain of memory definitions in programs. This helps us efficiently scan instructions. And `MemorySSA` can be seen as a virtual IR. It maps to memory accesses (instructions) in the program and maintains their interactions with the memory.

One may have noticed that in Section 3.3, Algorithm 7, when comparing generation numbers of `Inst` and `InVal`, `EarlyCSE` uses a function `isSameMemGeneration`. It might seem strange to use a separate function for simply comparing two numbers. We omitted this function in Section 3.3 because it was irrelevant to our methods. And it makes use of MemorySSA. We show it here as Algorithm 22.

Algorithm 22: EarlyCSE::isSameMemGeneration

Input: *EarilierInst, LaterInst*
if *EarlierInst has the same generation as LaterInst* **then**
  return *true*
EarlierMA ← Memory access of *EarlierInst*
LaterDef ← clobbering memory access of *LaterInst*
if *LaterDef dominates EarlierMA* **then**
  return *true*
return *false*

It first, certainly, checks the generation numbers of two instructions. However, it does not immediately return false when finding they are not equal. Instead, it makes use of MemorySSA to find the clobbering memory access of `LaterInst` and the memory access of `EarlierInst`. Since we know that both `EarlierInst` and `LaterDef` dominate `LaterInst`, if `LaterDef` dominates `EarlierInst`, then there cannot be any clobber between `EarlierInst` and `LaterInst`. In this case, we can safely do the CSE optimization. As an example, take a look at the following code:

```
x = 2;       // Clobber of LaterInst
r0 = x;      // <- EarlierInst
y = 1;
r1 = x;      // <- LaterInst
```

Now `LaterInst` and `EarlierInst` read from the same location. We try to know whether `LaterInst` is redundant. So we find the clobber of `LaterInst`, which is `x = 2;`. And `x = 2;` turns out to dominate `EarlierInst`, `r0 = x;`. This means that there is no clobber between `EarlierInst` and `LaterInst`, and indeed there is not. And we remove `r1 = x;` safely. This has shown that `MemorySSA` is a powerful tool. And we will also use it in the DSE pass.

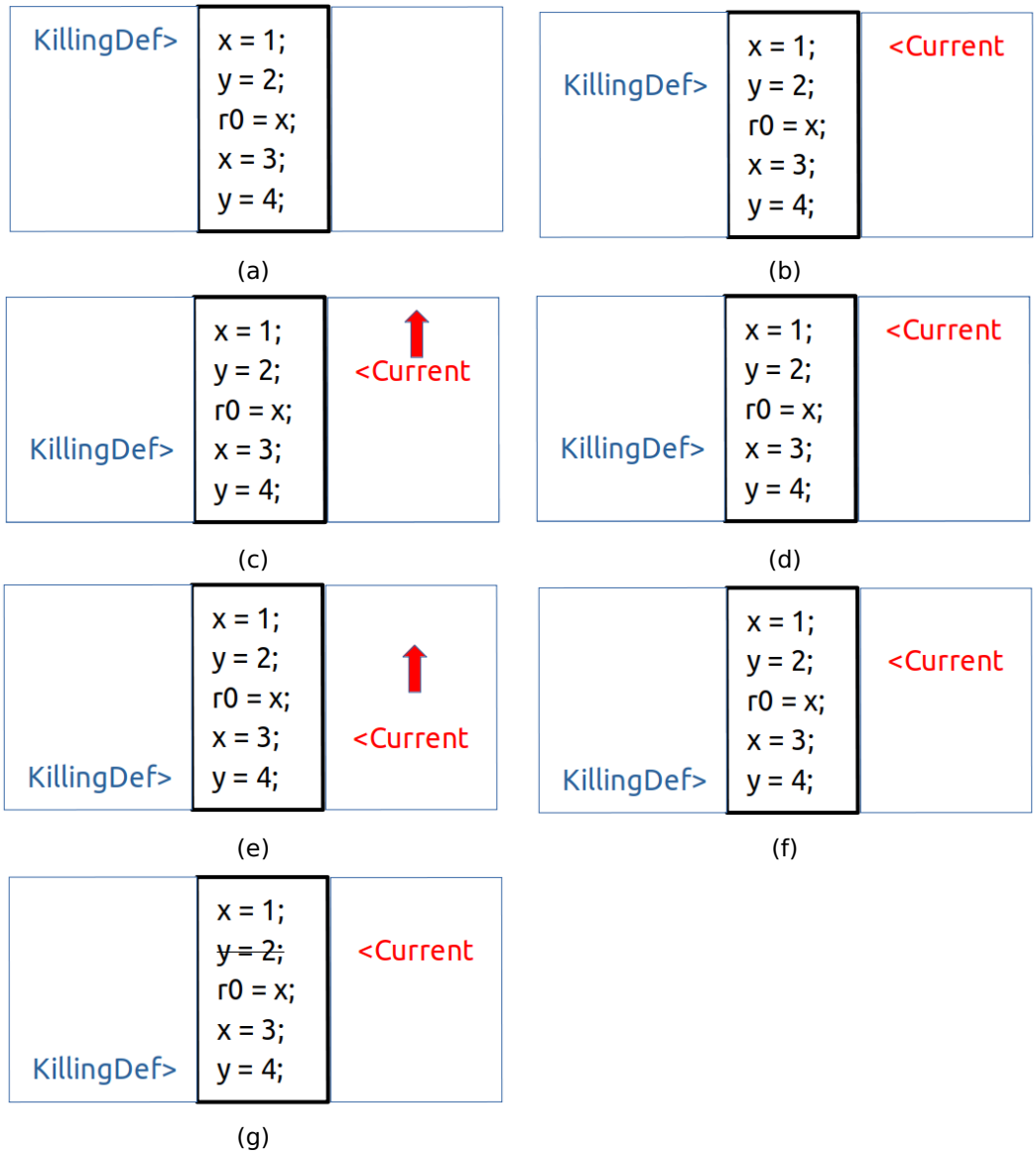


Figure 11: Process of DSE Pass Optimizing a Non-atomic Example

6.1.3 LLVM DSE

Now we can give an overall algorithm for the DSE pass. Consider the example in Figure 11:

- Figure 11a shows the initial state. `KillingDef` is the iterator over the instructions. It scans over `MemoryDef`'s. Every memory definition can potentially kill another one. `KillingDef` scans instructions in order, and tries to find OW patterns. `x = 1;` is a store but is the first definition in the current program. It could kill nothing, so we just skip.
- As Figure 11b, now we come to `y = 2;`. The defining access of it is `x = 1;`. We use the pointer, `Current` to walk upwards, to find stores killed by `KillingDef`. However, `x = 1;` and `y = 2;` write to unrelated locations. So nothing will happen.
- Next MemoryDef is `x = 3;`. And we start with its defining access, `y = 2;`, and go upwards as Figure 11c. Again, `y = 2;` writes to a different loca-

tion, so we continue going up. Then we find `x = 1;` as Figure 11d. Now `KillingDef` and `Current` write to the same location, which means `x = 3;` can possibly kill `x = 1;`. After confirming the location, we will check the uses of `x = 1;`. We find there is read `r0 = x;` that reads from `x`, which means the write `x = 1;` is used by another instruction and is not dead. So we cannot remove `x = 1;`.

- And finally `KillingDef` visits `y = 4;` as Figure 11e. First, `x = 3;` writes to a different location, so we skip it. Then we find `y = 2;` as Figure 11f. We check the uses of `y = 2;`, and we found that it is never used. So we remove it as Figure 11g.

This example gives an idea about the algorithm of the DSE pass. We will come to more details and our proposed atomic optimizations in the following parts.

6.2 Current Approach of DSE Pass

In this part, we describe algorithms used in DSE pass and explain how it forbids atomics.

6.2.1 Algorithm Description InstCombine

Algorithms used in DSE are shown from Algorithm 23 to Algorithm 28.

Algorithm 23: DSE

```
   Input: Function
1  for KillingDef in MemoryDefs of Function do
2      ToCheck ← [ ]
3      KillingLoc ← location of KillingDef
4      KillingUndObj ← object accessed by KillingDef
5      Insert Defining access of KillingDef into ToCheck
6      for Current in ToCheck do
7          MaybeDeadAccess ←
             getDomMemoryDef(KillingDef, Current, KillingLoc, KillingUndObj)
8          if MaybeDeadAccess is null then
9              continue
10         Insert Defining access of MaybeDeadAccess into ToCheck
11         OR ← Overwriting result of KillingDef over MaybeDeadAccess
12         if OR is Partial Earlier With Full Later then
13             Merge KillingDef into MaybeDeadAccess
14         if OR is KillingDef completely overwrites MaybeDeadAccess then
15             Remove MaybeDeadAccess
```

We can see that these algorithms have quite clear structures. As we have discussed the working procedure of DSE in the previous part, it might be unnecessary to explain the detailed process of these algorithms. We introduce these algorithms below. We marked the barriers against atomics as red, and we will discuss these barriers also.

Algorithm 24: getDomMemoryDef

Input: *KillingDef, StartAccess, KillingLoc, KillingUndObj*

```
Current ← StartAccess
for ;;Current ← Defining access of Current do
    if Current is liveOnEntry then
        return null
    if isDSEBarrier(KillingUndObj, Current) then
        return null
    if isReadClobber(KillingLoc, Current) then
        return null
    if !isRemovable(Current) then
        continue
    if Current does not write to the same object as KillingDef then
        continue
    break
Worklist ← [ ]
CurrentLoc ← location of Current
Push uses of Current into Worklist
for Use in Worklist do
    if isReadClobber(CurrentLoc, Use) then
        return null
return Current
```

Algorithm 25: isDSEBarrier

Input: *KillingUndObj, DeadI*

```
if DeadI is atomic then
    if DeadI has an ordering stronger than relaxed then
        return true
return false
```

Algorithm 26: isReadClobber

Input: *DefLoc, UseInst*

```
if UseInst is store then
    if UseInst has an ordering stronger than relaxed then
        return true
    return false
if !mayReadFromMemory(UseInst) then
    return false
return mayRefer(UseInst, DefLoc)
```

Algorithm 27: isRemovable

Input: *Inst*

```
if Inst is store then
    if Inst is ordered then
        return true
return false
```

Algorithm 28: mayRefer

```
   Input: Inst, Loc
 1 if Inst is ordered then
 2     return true
 3 if Inst is store then
 4     return false
 5 if Loc is not null and Inst is load then
 6     AR ← Alias Analysis Result of Inst at Loc
 7     if AR is no alias then
 8         return false
 9     if AR is must alias then
10         return true
11 return true
```

- DSE, as shown in Algorithm 23 is the entry of the pass. It iterates over the input function with the iterator `KillingDef`, and tries to find and eliminate memory definitions killed by `KillingDef`. In each iteration, it maintains a vector `ToCheck`, consisting of instructions possibly killed by `KillingDef`. Initially, `KillingDef` has only the defining access of `KillingDef` in it. And `getDomMemoryDef` will return the dead access killed by `KillingDef` if it finds any. And after that, defining access of the dead access will be inserted into `ToCheck` for the next check. This is to optimize the case like:

```
1     a = 1;  // <- Also Dead Accesses
2     a = 2;  // <- Dead Accesses
3     a = 3;  // <- KillingDef
```

  If `a = 3;` is the `KillingDef`, it kills `a = 2;` in the original `ToCheck` list. And `a = 1;` will be inserted into `ToCheck` after `a = 2;` is found. So that a series of dead stores could be killed. Note that the case above is for explanation. In practice, EarlyCSE will optimize this pattern before DSE.

  One thing interesting about `DSE` is the part starting from Line 11. It checks the overwriting relationship between `KillingDef` and `MaybeDeadAccess`. If `KillingDef` just overwrites a part of `MaybeDeadAccess`, like:

```
1     a = 0x0000;  // <- Dead Accesses
2     a_lower = 0xff;  // <- KillingDef
```

  where `KillingDef` only rewrites lower bits of `a`. We can merge the two writes to be:

```
1     a = 0x00ff; // <- Dead Accesses
```

  And if `KillingDef` completely overwrites `MaybeDeadAccess`, like the dummy examples we have shown, we simply remove `MaybeDeadAccess`.

- Then `getDomMemoryDef`, the key function of DSE, as shown in Algorithm 24. It does the critical job here: search upwards from the starting instruction, and return the dead access if it finds any. And to do this, it first walks along the chain of `MemoryDef`'s from the current access. If there is any clobber in the way, it returns null. And if it meets some instruction that is not removable, it skips that instruction.

If after the iteration starting from Line 2, we find a `Current` that could be dead, we push all uses of `Current` into `WorkList`. Then we check if any of these uses would read from `CurrentLoc`. If the checks find nothing, `Current` is possibly killed by `KillingDef`, and will be returned to `DSE`.

- The following algorithms are helper functions used by `getDomMemoryDef`, and they are the barriers against atomics here. `isDSEBarrier` is designed to avoid reordering any atomic instruction with an ordering stronger than relaxed (which is just working against us).

  `isReadClober` checks if `UseInst` may read from `DefLoc`. And it is this function that finds `r0 = x;` is a clobber in Figure 11. However, a store instruction stronger than relaxed will be taken as read from `DefLoc`. And on the last line of Algorithm 26, `mayRefer` is invoked. This is the load counterpart of `mayModify`. It is a part of LLVM Alias Analysis, and it determines whether `UseInst` may refer to `DefLoc`. And similar to `mayModify`, it assumes any ordered instruction may read from all memory locations, no matter if it is a read or a write, as shown in Algorithm 28.

  Finally `isRemovable`, which simply judges if `Inst` can be removed. And apparently, an ordered store is not removable for it.

6.3 Proposed Approach for DSE

We listed our modified algorithms, shown in Algorithm 29 to Algorithm 34. We marked the modified or newly added parts as cyan, and we suffixed our methods with "OA".

Algorithm 29: DSEOA

```
   Input: Function
1  for KillingDef in MemoryDefs of Function do
2      ToCheck ← [ ]
3      KillingLoc ← location of KillingDef
4      KillingUndObj ← object accessed by KillingDef
5      Insert Defining access of KillingDef into ToCheck
6      for Current in ToCheck do
7          MaybeDeadAccess ←
             getDomMemoryDefOA(KillingDef, Current, KillingLoc, KillingUndObj)
8          if MaybeDeadAccess is null then
9              continue
10         Insert Defining access of MaybeDeadAccess into ToCheck
11         OR ← Overwriting result of KillingDef over MaybeDeadAccess
12         if OR is Partial Earlier With Full Later then
13             Merge KillingDef into MaybeDeadAccess
14         if OR is KillingDef completely overwrites MaybeDeadAccess then
15             Remove MaybeDeadAccess
```

Algorithm 30: isDSEBarrierOA

```
Input: KillingUndObj, DeadI
if DeadI is atomic then
    if DeadI is load then
        return false
    if DeadI has an ordering stronger than relaxed then
        return true
return false
```

Algorithm 31: isReadClobberOA

```
Input: DefLoc, UseInst
if !mayReadFromMemory(UseInst) then
    return false
return mayReferOA(UseInst, DefLoc)
```

Algorithm 32: isRemovableOA

```
Input: Inst
if Inst is store then
    return true
return false
```

Algorithm 33: mayReferOA

```
Input: Inst, Loc
if Inst is store then
    return false
if Loc is not null and Inst is load then
    AR ← Alias Analysis Result of Inst at Loc
    if AR is no alias then
        return false
    if AR is must alias then
        return true
return true
```

For `DSE`, we just replaced `getDomMemoryDef` with our new version. And for helper functions used by `getDomMemoryDef`, we adjusted them and removed barriers of atomics, as we did in previous passes.

We made relatively more modifications to `getDomMemoryDef` since we need to enforce reordering constraints here. To be more specific, we added two flags: `PastSCRead`, which indicates whether we have gone through a sequentially consistent load, and `SeenAReleaseWrite`, which indicates whether we have seen a release store. And we added checks based on these flags which may control the iterations. These flags are added to enforce the rules in Table 2. we copied this table from Section 2, and marked the corresponding rules as red.

Algorithm 34: getDomMemoryDefOA

```
   Input: KillingDef, StartAccess, KillingLoc, KillingUndObj
 1 Current ← StartAccess
 2 PastSCRead ← false
 3 SeenAReleaseWrite ← false
 4 for ;;Current ← Defining access of Current do
 5     if Current is liveOnEntry then
 6         return null
 7     if SeenAReleaseWrite then
 8         return null
 9     if isDSEBarrierOA(KillingUndObj, Current) then
10         if Current is store then
11             SeenAReleaseWrite ← true
12         else
13             return null
14     if Current is load and has an ordering of Sequentially Consistent then
15         PastSCRead ← true
16     if Current is store then
17         if Current has an ordering of Sequentially Consistent and
            PastSCRead is true then
18             continue
19         if Current an ordering stronger than KillingDef then
20             continue
21     if isReadClobberOA(KillingLoc, Current) then
22         return null
23     if !isRemovableOA(Current) then
24         continue
25     if Current does not write to the same object as KillingDef then
26         continue
27     break
28 Worklist ← [ ]
29 CurrentLoc ← location of Current
30 Push uses of Current into Worklist
31 for Use in Worklist do
32     if isReadClobberOA(CurrentLoc, Use) then
33         return null
34 return Current
```

$\downarrow a \backslash b \rightarrow$	$R_{NA\|RLX\|ACQ}(l')$	$R_{SC}(l')$	$W_{NA}(l')$	$W_{RLX}(l')$	$W_{REL\|SC}(l')$
$R_{NA}(l)$	✓	✓	✓	✓	✗
$R_{RLX}(l)$	✓	✓	✓	✓	✗
$R_{ACQ\|SC}(l)$	✗	✗	✗	✗	✗
$W_{NA\|RLX\|REL}(l)$	✓	✓	✓	✓	✗
$W_{SC}(l)$	✓	✗	✓	✓	✗

Table 2: Allowed Reorderings $a; b \rightsquigarrow b; a$, assuming locations $l \neq l'$ (Copy)

7 Evaluation

In this section, we evaluate our compiler in two forms:

- Randomly generated memory accesses. We use random tests of different types to check the effectiveness of our compiler. Also, we measured the compile time to see the overhead of enabling our optimizations.
- Concurrent benchmarks. We build concurrent applications with our compiler and see if we get any performance gain or code size change.

7.1 Randomly Generated Memory Accesses

First, we used a random test case generator [3] to generate random memory accesses as tests. In each test case, the generator generates a function consisting of memory accesses. These accesses cover reads and writes, and have all types of memory orderings from non-atomic to sequentially consistent. And they can be divided into five categories:

- **a** Straight-Line test cases. The function has only one basic block.
- **b** Branches tests cases. The function has conditional branches.
- **c** Dead paths test cases. The function has empty basic blocks.
- **d** Loops test cases. The function has loops.
- **e** Mixed test cases. The function may have all structures listed above.

An example of type **e** is shown below:

```
#include <atomic>
using namespace std;
atomic<int> x,y;
int a0, a1, a2;
int func(bool flagloop, bool flag0, bool flag1,  int dummy) {
    int rx=-20, ry=-23, old = 23, nw = 43, r=50;
    int r0=-45, r1=-43, r2=-14;
    r += a0;
    y.store(20,memory_order_release);
    r += a0;
    r += a2;
    y.store(20,memory_order_release);
    if(flag0) {
    } else {
    a2 = 2;
    }
    r += a2;
    a0 = 0;
    ry += y.load(memory_order_acquire);
    r += a2;
}
```

Just as in this example, test cases may have accesses with different orderings to shared atomic variables like `x`, shared non-atomic variables like `a0`, and local variables like `r`.

We implemented our optimizations as optional functions of the compiler enabled using command-line arguments. For example, to allow EarlyCSE to optimize atomics, use `clang++ ^your_configurtions -mllvm -cse-optimize-atomic`. We generated 100 test cases of each type mentioned above, and compiled them using the following:

- `clang++ -O3` The original clang compiler with optimization level `O3`.

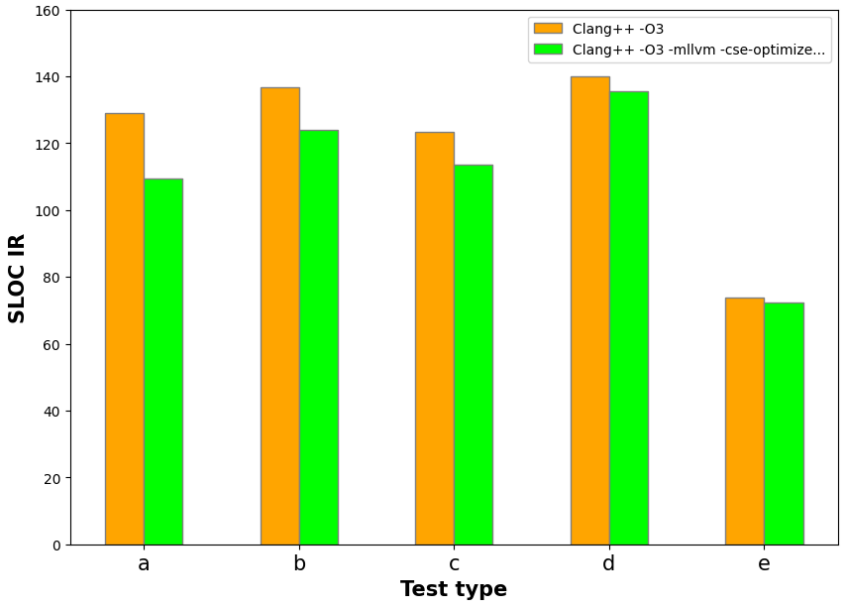


Figure 12: IR File Size Change After Optimizing Atomics

- `clang++ -O3 -mllvm -cse-optimize-atomic -mllvm -ic-optimize-atomic -mllvm -dse-optimize-atomic` Set the flags enabling optimizations we implemented besides level `O3`.

We counted the average of output IR files' source lines of code (SLOC) in each type. The results are shown in Figure 12. This shows that a number of instructions, which current LLVM cannot remove using its highest optimization level, can be further optimized by our compiler. And it can be observed in the figure that, the more complex the test cases are (from a to e), the fewer optimizations are done. For Straight-Line test cases (a), we can remove 15% lines compared to clang-14, while for mixed tests (e), only 2% can be further removed.

This is because in simple test cases like straight-line tests, instruction reordering constraints are only from the current basic block. However, in tests with branches or loops, additional constraints can also be added from other basic blocks. Therefore, there are fewer optimization opportunities in complex cases.

We also counted the number of accesses removed by each pass, as shown in Figure 13. The first thing to note is that the InstCombine pass can rarely remove instructions compared with the other two. This is in fact a result of using MemorySSA in EarlyCSE, as we mentioned in Section 6.1.2. EarlyCSE becomes quite powerful in removing redundant loads with the help of MemorySSA and left little work to InstCombine. Although EarlyCSE does not use alias analysis which is deemed to be time-consuming, MemorySSA enabled it to find RAR and RAW patterns efficiently. Another observation is that for more complex cases (c-e), EarlyCSE has much less to do, while DSE dominates the optimizations. This is in line with the original intention of EarlyCSE: to remove trivially dead instructions in simpler cases. Also, this shows that EarlyCSE does help offload other passes. It takes responsibility when it is able to.

To see the overhead of these optimizations, we measured the compile time of these test cases, shown in Figure 14. Overall, the compile time overhead is negligible as we expected. And complex cases may take a longer time, but still, the difference is under 1%. And this is one of the main benefits of our design. Adapting existing algorithms could minimize the impact on compile time.

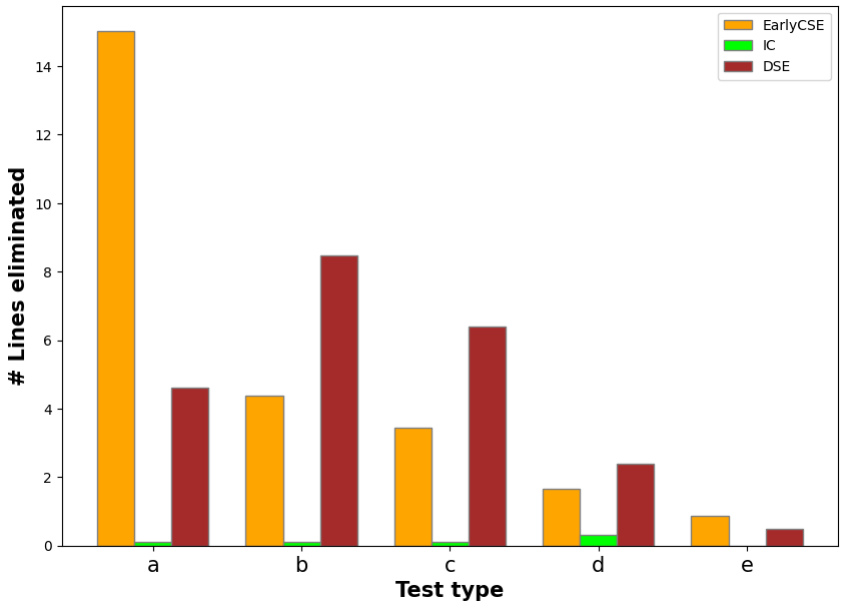


Figure 13: Numbers of lines eliminated by each pass

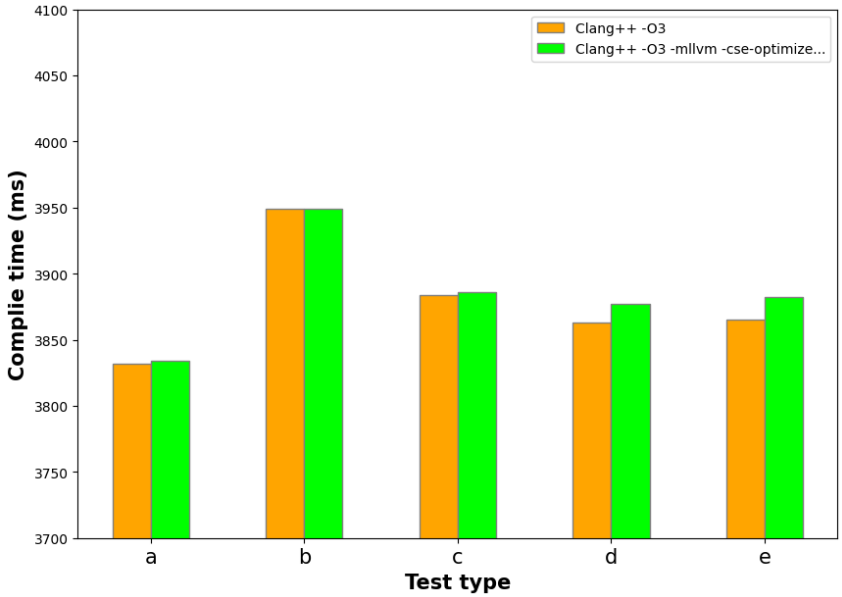


Figure 14: Compile Time Change After Optimizing Atomics

7.2 Benchmarking

We also benchmarked our work with several software making use of C/C++ atomics. We built the following applications using both LLVM clang-14 and our compiler, and compared the generated IR files to check the difference:

- **CDSChecker** A tool for exhaustively exploring the behaviors of concurrent code under the C/C++ memory model [11]. The name CDS stands for Concurrent Data Structures, which are implemented using atomics. It uses several techniques for modeling the behaviors of the relaxed memory model and can be used to unit test concurrent data structure implementations.
- **iris** An asynchronous logging library. It uses a "background thread" (the logging thread) to help offload logging from other threads. And other threads will keep a lock-free queue to collect messages. The logging thread will scan these queues to collect messages and write them into the log file. In this way, it can minimize the overhead of logging messages. Maintaining these queues will involve atomic memory accesses.
- **Silo** An in-memory database system [16]. It aims to achieve high performance as well as scalability on modern multicore processors. Silo was designed from the ground up to use system memory and caches efficiently. Databases are a scenario where atomic accesses can be widely used.
- **Mabain** A light-weighted C++ key-value store library. It saves data on disk but allows users to specify how much data can be mapped to shared memory. It supports multi-Thread/multi-Process insertion/update, which is similar to database applications. And it makes use of atomic accesses.

We built all these applications with our compiler enabling optimizations of atomics. We compared the generated and optimized IR files with the ones produced by clang-14. And we observe no difference so far. In other words, our compiler does not find patterns to be optimized in these given concurrent applications. Since we did not change the backend of LLVM, the same IR files will result in the same binaries. So comparing executions will make no sense. We omitted the comparison here.

The reason for the result could be that these applications are carefully developed and left no chance for our compiler to be optimized. And after checking the source code, one can find that even in these concurrent applications, atomic memory accesses are only a small portion compared to the regular ones. In the current applications, atomic variables are typically used as signals to communicate between threads. They are not heavily used to access data like regular variables. And if these atomic instructions are properly used, there would be few chances for improvement.

Of course, we just tested several concurrent benchmarks. It is totally possible we could find opportunities if we try a wider range of applications.

8 Related and Future Work

8.1 Related Work

Besides the work mentioned in Section 2, on which our project is closely based, there are several more related works to mention here.

There are works on SC-preserving compilers. Sequentially Consistent is the most intuitive concurrent memory model. Under SC, the individual instructions will look as if executed in a global sequential order consistent with program orders in each thread. This a relatively simpler way to deal with ordering. And SC-preserving means that every SC behavior of a generated binary is guaranteed to be an SC behavior of the source program. Instead of investigating optimizing opportunities in relaxed models, SC-preserving compilers try to keep the SC memory model. For example, bulk compiler [1] and bulkSC hardware [2] together guarantee SC at the language level. This is a costly way since extra hardware is required. And it has been shown that [10] giving up chances in relaxed memory models and choosing an SC-preserving compiler does not hurt performance much as assumed. Works in this direction can be taken as attempts at another aspect.

On the other hand, recently there have been extended such SC-centric models incorporating relaxed atomics. Instead of the popular data-race-free-0 (DRF0) model which requires programmers to avoid data races, Matthew et al. [15] proposed Data-Race-Free-Relaxed (DRFrlx), that extends DRF0 to provide SC-centric semantics for the common use cases of relaxed atomics. According to their evaluation, there is little performance gain for most cases. However, in some cases, a significant performance gain is observed.

There are also works showing the potential of optimizing atomics. Weak memory models give the programmer a wide scope of choices about how to implement the exact inter-thread communication using the shared memory. These choices will have semantic and performance consequences. So it makes sense to find out the impact of these choices on performance. Carl et al. [14] defined techniques for evaluating the impact of various choices in using weak memory models, such as where to put fences, and which fences to use. And they provided techniques that help programmers understand the performance implications when identifying and resolving any semantic/performance trade-offs, which we may also take as advice when implementing compiler optimizations. And it has been shown in works [9] that proper use of barriers would bring obvious performance gain.

8.2 Future Work

We have implemented the optimizations of RAR, RAW, and OW for atomics in LLVM. And there are still several directions for the future work:

- **Try more aggressive optimizations.** In Section 4, we said we choose to implement atomic transformations based on existing passes. Nevertheless, this will lose some chances of optimizations. Ordering is an important feature of atomics. Optimizing non-atomic memory accesses does not raise the ordering problem, so one can arbitrarily choose a "direction" of reordering instructions. For now, in the passes we have seen, LLVM is always (tentatively) moving the dead instruction towards the killing instruction, and eliminating the dead one if possible. This will keep the original relative order between instructions, and this seems to be a good choice for non-atomic cases.

  However, as the example, we showed in Section 4.2, trying a different way of reordering may introduce new opportunities. Following existing passes

will not allow these chances. So building new passes dedicated to atomics could be an idea.

- **Validate the compiler.** Software validation, especially for infrastructures like compilers, is always an important topic [8]. We believe our compiler optimizing the atomics is correct because we made our modifications and implemented reorderings based on existing proven theories. However, if we wish to eventually implement this work on LLVM, validating/formally proving the correctness of the compiler would be a good thing to do.
- **Try more atomic objects.** We have worked on atomic reads and writes. Still, there are other atomics to be optimized, including fences, atomic read-modify-write, etc. Fences, for example, have been thought of as an important performance overhead, Covering these atomic instructions may bring further improvements and could make the work more complete.

9 Conclusion

This thesis project builds an LLVM-based compiler optimizing atomic memory accesses. Weak memory models are designed to enable a wide range of optimizations for concurrent programs and have been used by modern microprocessors. As a state-of-the-art compiler, LLVM supports weak memory models but disallows most possible optimizations over atomics. We carefully found the algorithms concerning classic elimination optimizations over memory accesses, removed the barriers that forbid optimizations of atomics, and added constraints to ensure the soundness of our modifications.

We implemented our optimizations as an optional function. One can turn on the optimizations using `-mllvm` flags. And when not set, our compiler works exactly the same as LLVM 14.0.0.

For a comprehensive conclusion, we would like to give answers to questions in Section 1.2:

- The current LLVM is conservative. In the passes we worked through in this project, almost all reordering, modification, or elimination of atomic accesses are disallowed. In fact, to ensure no atomic operation can escape, the notion of avoiding optimizing atomics is deeply rooted in the whole LLVM project, from core libraries like alias analysis to the specific implementation of optimizing passes. One of the reasons might be that atomic instructions often have ordering constraints. And reordering or deleting them might be risky.

  However, different passes in LLVM behave differently. For example, DSE allows basic optimizations for relaxed atomic accesses. Such differences might be because as a large open-source project, LLVM is maintained by developers across the world. Developers responsible for every part may have slightly different choices.

- However, optimizing atomics is possible. It has been proven that reordering and eliminating atomic instructions under the given constraints are safe. As mentioned in Section 2.4, there are two types of rules to obey: elimination rules, which provide the patterns we can optimize, and reordering rules, which give constraints about reordering ordered instructions.

- To enable these optimizations, we have several design choices. And the one we finally chose, is to modify existing passes to adapt optimizations for atomics. We chose this method because in this way we can better reuse existing algorithms. What's more, since no new pass is required and the original structures of the passes are preserved, our modifications will not bring additional complexity to the algorithms.

  And to actually implement these optimizations, we need to first find the passes taking care of our interesting patterns. Then we need to look into these passes and remove the barriers/checks which forbid atomic accesses from being processed. And finally, we shall add constraints, and adapt them to existing algorithms of passes, to make our transformations safe.

- We tested our compiler by using it to optimize randomly generated memory accesses. Compared to the original LLVM compiler, we can move a significant portion (2% to 15% in our test cases) of redundant atomic instructions, while not introducing an obvious overhead in compile time. Cases with more complex control flows have smaller spaces for optimizations. And we also evaluated our compiler with some concurrent benchmarks. However, we do not observe apparent performance gains yet. The reason could be that these concurrent applications do not contain patterns we optimize.

And in the future, this work could be further extended to cover more atomic instructions. Since ordering is a unique feature of atomics, it is also possible to develop a new pass to perform more aggressive optimizations on atomic accesses. Maybe more performance gains could be achieved when these works are done.

References

[1] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong. Bulkcompiler: High-performance sequential consistency through cooperative compiler and hardware support. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, page 133144, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587981. doi: 10.1145/1669112.1669131. URL https://doi.org/10.1145/1669112.1669131.

[2] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. Bulksc: Bulk enforcement of sequential consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 278289, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937063. doi: 10.1145/1250662.1250697. URL https://doi.org/10.1145/1250662.1250697.

[3] S. Chakraborty and V. Vafeiadis. Validating optimizations of concurrent c/c++ programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 216226, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450337786. doi: 10.1145/2854038.2854051. URL https://doi.org/10.1145/2854038.2854051.

[4] S. Chakraborty and V. Vafeiadis. Formalizing the concurrency semantics of an llvm fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 100110. IEEE Press, 2017. ISBN 9781509049318.

[5] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. *SIGPLAN Not.*, 33(5):106117, may 1998. ISSN 0362-1340. doi: 10.1145/277652.277670. URL https://doi.org/10.1145/277652.277670.

[6] J. K. Kuderski. Dominator trees and incremental updates that transcend time. URL https://llvm.org/devmtg/2017-10/slides/Kuderski-Dominator_Trees.pdf.

[7] C. Lattner. The architecture of open source applications (vol 1) : Llvm. URL https://aosabook.org/en/v1/llvm.

[8] X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. *SIGPLAN Not.*, 41(1):4254, jan 2006. ISSN 0362-1340. doi: 10.1145/1111320.1111042. URL https://doi.org/10.1145/1111320.1111042.

[9] N. Liu, B. Zang, and H. Chen. No barrier in the road: A comprehensive study and optimization of arm barriers. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 348361, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368186. doi: 10.1145/3332466.3374535. URL https://doi.org/10.1145/3332466.3374535.

[10] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A case for an sc-preserving compiler. *SIGPLAN Not.*, 46(6):199210, jun 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993522. URL https://doi-org.tudelft.idm.oclc.org/10.1145/1993316.1993522.

[11] B. Norris and B. Demsky. Cdschecker: Checking concurrent data structures written with c/c++ atomics. *SIGPLAN Not.*, 48(10):131150, oct

2013. ISSN 0362-1340. doi: 10.1145/2544173.2509514. URL https://doi.org/10.1145/2544173.2509514.

[12] L. Projec. The llvm compiler infrastructure project. URL https://www.llvm.org/.

[13] L. Project. Llvm documentation: Memoryssa, 2023. URL https://llvm.org/docs/MemorySSA.html.

[14] C. G. Ritson and S. Owens. Benchmarking weak memory models. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP Principles and Practice of Parallel Programming. ACM, New York, USA, February 2016. URL https://kar.kent.ac.uk/51638/.

[15] M. D. Sinclair, J. Alsop, and S. V. Adve. Chasing away rats: Semantics and evaluation for relaxed atomics on heterogeneous systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 161174, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348928. doi: 10.1145/3079856.3080206. URL https://doi.org/10.1145/3079856.3080206.

[16] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 1832, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522713. URL https://doi.org/10.1145/2517349.2522713.

[17] 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 *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 209220, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2676995. URL https://doi-org.tudelft.idm.oclc.org/10.1145/2676726.2676995.