Hyeongyu Kim
gusrb406@snu.ac.kr
Seoul, South Korea
Timezone: Korea Standard Time (UTC+9)

Fix miscompilation issues in LLVM IR using the ‘Freeze’ instruction

GSoC Project proposal for LLVM

Synopsis

A. Introduction

LLVM performs various optimizations to increase performance. However, optimizations have been the cause of multiple miscompilations as well. It is difficult to determine whether an optimization is correct because bugs occur in corner cases and it’s hard to test them all for each optimization. But, thanks to Alive2[1], we can formally check the correctness of optimizations. Alive2 already found more than 50 miscompilation bugs in LLVM.

Many of the bugs found are related to the notion of undef and poison values in LLVM. The semantics of these values has been formalized recently, and existing optimizations are being fixed to match the semantics. For example, branching on undef is now defined as undefined behavior (UB), and several branch-related optimizations are being fixed.

However, fixing some of the optimizations requires further work, mainly due to performance concerns. One representative example is loop unswitch: it is incorrect, but still unfixed because fixing it causes big regressions in some benchmarks. Our goal is to

[1] https://github.com/aliveToolkit/alive2

fix such miscompilations with minimal-to-no performance regression. This project will investigate what is necessary to recover slowdowns and upstream the patches to LLVM.

B. Inconsistency in LLVM

First, we'll explain why loop unswitch is incorrect and how it can be fixed. It is well known that there is an inconsistency between global value numbering (GVN) and loop unswitching (LU)[2]. GVN requires branching on undef to be UB as described above, but LU is incorrect in this semantics. For brevity, we'll omit why GVN requires branching on undef to be UB, and explain the LU case only.

```
while (c) {
  if (c2) { foo }
  else    { bar }
}
```

```
if (c2) {
  while (c) { foo }
} else {
  while (c) { bar }
}
```

LU transforms the left code to the right if c2 is a loop-invariant expression and hoisting the conditional branch is beneficial. However, this isn't safe due to the existence of UB. If c is false and c2 is poison, the original program (left) is well-defined. But, after the transformation (right), branching on poison is executed regardless of the value of c, which is UB. This became the source of miscompilations when combined with GVN[3]. It was the source of a bootstrap failure in 2017 as well[4].

To solve this kind of branch-on-undef problem, a new instruction "freeze" was added to LLVM. With the freeze instruction, we can prevent propagation of undef and poison values. For the loop unswitch example above, using freeze(c2) instead of c2 is the right fix because it prevents branching on poison (which is UB). So far, freeze has already been used to fix a few bugs[5].

Loop unswitch was also fixed using freeze in the past, but the patch was reverted due to performance issues, and so the miscompilation problems still exist today.

---

[2] "Taming undefined behavior in LLVM" from https://sf.snu.ac.kr/publications/undefllvm.pdf.
[3] "LLVM bugs PR27506, PR31652":https://bugs.llvm.org/show_bug.cgi?id=27506, https://bugs.llvm.org/show_bug.cgi?id=31652
[4] https://lists.llvm.org/pipermail/llvm-dev/2017-July/115580.html
[5] "D76179, D76483 issue": https://reviews.llvm.org/rG6ad63606ea4afde9043148509449ab984bfd499a, https://reviews.llvm.org/rG49f75132bcdfcbc23010252e04e43fe0278ae1e7

C. Expected cause of performance loss

There are two possible causes for the performance degradation of Freeze. The first problem is that existing optimizers are not optimizing freeze instruction to the appropriate form. The second problem is that freeze instruction is redundantly inserted sometimes. We'll cover the details of these two problems and their solutions.

Project Goal

In this GSoC project, I will solve the branch-on-undef problem by fixing the miscompilations using freeze and resolving the performance degradation caused by the use of freeze.

There are several sources of performance regressions. One is that the information that certain values cannot be undef/poison in the C/C++ source code is lost after conversion to LLVM IR. I plan to solve the problem by obtaining more information about non-undef/poison information from the C/C++ source code.

Second, optimizations such as SimplifyCFG, InstCombine, and InstSimplify don't consider the freeze instruction. Therefore, optimizations often fail when freeze is used. For example, "freeze(%a) & %a" can be replaced with "freeze(%a)," but LLVM does not perform this currently. Other more complex optimizations such as vectorization will also need fixing. An increase in performance is expected if the freeze instruction is handled by the vectorization algorithm.

Third, we can also improve LLVM's ValueTracking to check whether a value is neither undef or poison for more complex expressions.

In short, I will do the following things to make LLVM better.

- Fix the miscompilation issues related to branch-on-undef using the freeze instruction.
- Demonstrate that freeze instruction solves the miscompilation problems and quantitatively analyze the performance loss, compiled time, binary size difference caused by the use of freeze.

  Fix and/or add new optimizations to reduce (or even eliminate) the performance loss due to the use of freeze.

Approach

- Implement straightforward fixes for the branch-on-undef issues with freeze.
    - Will fix loop unswitch and simplifyCFG.
- Check for unnecessary freeze instructions in the output.
    - Analyze the sample programs given when patches using freeze were reverted in the past.
    - Compile the LLVM testsuite and check diff of assembly code.
- Use a standard method to calculate the performance, compile time, the code size difference between patches.
    - I will use standard benchmark programs like SPEC CPU2017 and LLVM's test suite.
- Optimize performance by getting more information from C/C++ source code.
    - Create a C/C++ sample program where an inefficient freeze is inserted.
    - Figure out the information I need to get from the C/C++ frontend and discuss it with other LLVM/clang developers.
- Optimize performance by improving the LLVM IR optimizer that already exists.
    - Improve aspects where existing optimizers do not handle freeze instructions.
    - Write a test program where optimization must occur.
    - Improve SimplifyCFG, InstCombine, and InstSimplify optimizers to optimize away freeze instructions.
- After making these improvements, compare the performance loss, compiled time, binary size difference before and after the project.
    - Performance comparison of the compiled program is essential.
    - Compare the compile-time and compiled program's size to see if there is any noticeable change.
    - Summarize and visualize the changes.

Deliverables

- Patches for the open branch-on-undef issues like loop unswitch and simplifyCFG.
- Patches for clang to make it produce more information about values that are guaranteed to be non-undef/poison (if needed).
- Freeze-aware version of SimplifyCFG, Instcombine, and InstSimplify optimizer and test programs for them.
- Charts to compare performance differences between freeze and non-freeze versions of LLVM.

Timeline

Period	Task
Application Review Period [April 14 - May 18]	- Search for more information about UB, undef, and poison. - Get used to C's specification.
Community Bonding Period [May 18 - June 8]	- Get involved with the LLVM community. - Get a working knowledge of interaction between clang and LLVM IR.
Coding Starts	
Week 1 and 2 [June 8 - June 22]	- Begin implementation of straightforward fixes for the branch-on-undef issues in loop unswitching. - Measure performance loss due to the use of freeze. - Find the cause of performance loss by diffing assembly code of LLVM's test suite. - Check performance difference by CPU SPEC2017.
Week 3 and 4 [June 22 - July 6]	- Create a C/C++ sample program where an inefficient freeze is inserted and find a way to avoid it. - Improve clang to expose information needed to optimize away freeze instructions.
Week 5 and 6 [July 6 - July 20]	- Discuss with LLVM/clang developers about added information that was proposed in the previous week. - Based on the discussion, improve implementation. - Test the difference and make a report for mid-term evaluation.
Evaluation Periods	

[July 13 - July 17]	
Week 7 and 8 [July 20 - August 3]	- Improve the SimplifyCFG optimizer to make it freeze-aware.
Week 9 and 10 [August 3 - August 17]	- Improve Instcombine and InstSimplify to make them freeze-aware. - Write a test program to prove that it does not affect the behavior of the compiled program.
Final Evaluations [August 17 - August 24]	- Draw a graph to help understand changes in performance. - Organize the problems solved through the freeze and organize the performance difference caused by the freeze in an easily recognizable way.

About Me

I am an undergraduate student at the Seoul National University, and I am taking a compiler course by Professor Gil Hur this semester. It is a class that makes an LLVM IR optimizer for special hardware and compares the performance with other students. I became interested in LLVM during the course, and started to contribute to LLVM by fixing a miscompilation problem related to InstCombine (D99481). Previously, I worked as a C++ programmer in an antivirus software company. During this time, I was able to gain knowledge for large-scale program development using C++, which will significantly help me contribute to LLVM. GSoC will be an excellent opportunity to contribute, and I am confident in solving problems associated with branch-on-undef.