LifeJacket: Verifying Precise Floating-Point Optimizations in LLVM

Andres Nötzli Fraser Brown
Stanford University, USA
{noetzli,mlfbrown}@stanford.edu

Abstract

Users depend on correct compiler optimizations but floating-point arithmetic is difficult to optimize transparently. Manually reasoning about all of floating-point arithmetic's esoteric properties is error-prone and increases the cost of adding new optimizations.

We present an approach to automate reasoning about precise floating-point optimizations using satisfiability modulo theories (SMT) solvers. We implement the approach in LifeJacket, a system for automatically verifying precise floating-point optimizations for the LLVM assembly language. We have used LifeJacket to verify 43 LLVM optimizations and to discover eight incorrect ones, including three previously unreported problems. LifeJacket is an open source extension of the Alive system for optimization verification.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Correctness proofs

Keywords Verification, Floating-point arithmetic, SMT, LLVM

1. Introduction

Floating-point arithmetic is the de facto standard for representing operations on real numbers in code. Since programmers expect good running times from their programs, languages and compilers often recognize floating-point as a primitive type and optimize floating-point operations. Many of these optimizations are implemented as highly local transformations in the basic blocks of a program, transformations called peephole optimizations. When compiler developers create these optimizations, they have to carefully consider all possible inputs and make sure that the optimizations do not alter the meaning of programs.

Unfortunately, the developers' manual reasoning might be error prone. Figure 1 shows an example of an invalid transformation implemented in LLVM 3.7.1. We discuss the specification language in more detail in Section 3.2 but, at a high-level, the transformation simplifies $+0.0 - (-0.0 - x)$ to x, an optimization that is correct in the realm of real numbers. Because floating-point numbers distinguish between negative and positive zero, however, the optimization is not valid if $x = -0.0$, because the original code returns $+0.0$ and the optimized code returns -0.0. While the zero's sign may be insignificant for many applications, the unexpected sign change may cause a ripple effect. For example, the reciprocal of zero is defined as $1/+0.0 = +\infty$ and $1/-0.0 = -\infty$.

This example illustrates the difficulty [8] of wrestling with floating-point's unintuitive features (e.g. signed zeros, limited precision, rounding, special values, and non-associativity). While these properties may seem overtly complex, the IEEE 754-1985 standard and its stricter IEEE 754-2008 successor were carefully designed for non-expert users. Before the standardization, there were a wide range of incompatible floating-point implementations with occasionally bizarre properties: for example, one implementation provided numbers that were not equal to zero for comparisons but were treated as zeros for multiplication and division [14]. Despite advances in standardization, program correctness and reproducibility still rest on a fragile interplay between developers, programming languages, compilers, and hardware implementations.

Incorrect optimizations can silently corrupt programs by altering their semantics. Floating-point optimizations, though, may sometimes take advantage of the representation's inherent imprecision; their changes may negatively affect the precision of floating-point expressions. In fact, some compilers address the tension between speed and reproducibility by dividing floating-point optimizations into two groups, precise and imprecise optimizations, where imprecise optimizations are optional (e.g. the `-ffast-math` flag in `clang`). While precise optimizations retain the original semantics, imprecise ones produce reasonable results on common inputs (e.g. not for special values) but are arbitrarily bad in the general case. Our work focuses on precise optimizations because they are both more amenable to verification and arguably harder to get right.

We present LifeJacket, a tool that allows LLVM developers to verify precise floating-point optimizations, alleviating the challenge of manually reasoning about edge cases. LifeJacket builds on Alive [10], a tool for verifying LLVM optimizations, extending it with floating-point support. Our contributions are as follows:

- We describe the background for verifying precise floating-point optimizations in LLVM and propose an approach using SMT solvers.
- We implement the approach in LifeJacket[1], an open source fork of Alive that adds support for floating-point types, floating-point instructions, floating-point predicates, and certain fast-math flags.
- We validate the approach by verifying 43 optimizations. LifeJacket finds 8 incorrect optimizations, including three previously unreported problems in LLVM 3.7.1.

In addition to the core contributions, our work also lead to the discovery of issues in Z3 [7], the SMT solver used by LifeJacket, related to floating-point support.

[1] `https://github.com/4tXJ7f/alive`

```
Name: PR26746
%a = fsub -0.0, %x
%r = fsub +0.0, %a
  =>
%r = %x
```

Figure 1. Incorrect transformation involving floating-point instructions in LLVM 3.7.1.

2. Related Work

Alive is a system that verifies LLVM peephole optimizations. LifeJacket is a fork of this project that extends it with support for floating-point arithmetic. We are not the only ones interested in verifying floating-point optimizations; after our submission, two of the Alive authors independently announced Alive-FP, an extension of Alive with the same goal [6].

Our work intersects with the areas of compiler correctness, optimization correctness, and analysis of floating-point expressions.

Research on compiler correctness has addressed floating-point arithmetic and floating-point optimizations. CompCert, a formally-verified compiler, supports IEEE 754-2008 floating-point types and implements two floating-point optimizations [3]. In CompCert, developers use Coq to prove optimizations correct, while LifeJacket proves optimization correctness automatically. Darulova et al. [5] describe a technique to compile programs over real numbers into floating-point programs with guaranteed precision.

Regarding optimization correctness, researchers have explored both the consequences of existing optimizations and techniques for generating new optimizations. Recent work has discussed consequences of unexpected optimizations [16]. In terms of new optimizations, STOKE [13] is a stochastic optimizer that supports floating-point arithmetic and verifies instances of floating-point optimizations with random testing. Souper [1] discovers new LLVM peephole optimizations using an SMT solver. Similarly, Optgen generates peephole optimizations and verifies them using an SMT solver [4]. All of these approaches are concerned with the correctness of new optimizations, while our work focuses on the correctness of existing ones. Vellvm, a framework for verifying LLVM transformations using Coq, also operates on existing transformations but does not do automatic reasoning.

Researchers have explored estimating floating-point round-off errors [15] and improving the accuracy of floating-point expressions [11]. These efforts are more closely related to imprecise optimizations and provide techniques that could be used to analyze them. Lee et al. [17] describe a semi-automatic technique to verify mixed floating-point and bit manipulations at a machine code level. Z3's support for reasoning about floating-point arithmetic relies on a model construction procedure instead of naive bit-blasting [18].

3. Background

Our work verifies LLVM floating-point optimizations. These optimizations take place on LLVM assembly language, a human-readable, low-level language. The language serves as a common representation for optimizations, transformations, and analyses. Front ends (like `clang`) generate output in this language and back ends consume it to generate machine code for different architectures.

Our focus is verifying peephole optimizations implemented in LLVM's InstCombine pass which replaces small subtrees in the program tree without changing the control-flow graph. Alive verifies some InstCombine optimizations, but does not support optimizations involving floating-point arithmetic. Instead of building LifeJacket from scratch, we extend Alive with the machinery to verify floating-point optimizations. To give the necessary context for discussing our implementation (§ 4), we describe LLVM's floating-point types and instructions (§ 3.1) and give a brief overview of Alive (§ 3.2).

3.1 Floating-Point Arithmetic in LLVM

In the following, we discuss LLVM's semantics of floating-point types and instructions. This information is largely based on the LLVM Language Reference Manual for LLVM 3.7.1 [2] and the IEEE 754-2008 standard. For completeness, we note that the language reference does not explicitly state that LLVM floating-point arithmetic is based on IEEE 754. However, the language reference refers to the IEEE standard multiple times, and LLVM's floating-point software implementation `APFloat` is explicitly based on it.

Floating-point types LLVM defines six different floating-point types with bit-widths ranging from 16 bit to 128 bit. Floating-point values are stored in the IEEE binary interchange format, which encodes them in three parts: the sign s, the exponent e and the significand t. The value of a normal floating-point number is given by: $(-1)^s \times (1 + 2^{1-p} \times t) \times 2^{e-bias}$, where $bias = 2^{w-1} - 1$ and w is the number of bits in the exponent. The range of the exponents for normal floating-point numbers is $[1, 2^w - 2]$. Exponents outside of this range are used to encode special values: subnormal numbers, Not-a-Number values (`NaN`s), and infinities.

Floating-point zeros are signed, meaning that -0.0 and $+0.0$ are distinct. While most operations ignore the sign of a zero, the sign has an observable effect in some situations: a division by zero (generally) returns $+\infty$ or $-\infty$ depending on the zero's sign, for example. As a consequence, $x = y$ does not imply $\frac{1}{x} = \frac{1}{y}$. If $x = 0$ and $y = -0$, $x = y$ is true, since floating-point $0 = -0$. On the other hand, $\frac{1}{x} = \frac{1}{y}$ is false, since $\frac{1}{0} = \infty \neq -\infty = \frac{1}{-0}$.

Infinities ($\pm\infty$) are used to represent an overflow or a division by zero. They are encoded by setting $t = 0$ and $e = 2^w - 1$. Subnormal numbers, on the other hand, are numbers with exponents below the minimum exponent; normal floating-point numbers have an implicit leading 1 in the significand that prevents them from representing these numbers. The IEEE standard defines the value for subnormal numbers as: $(-1)^s \times (0 + 2^{1-p} \times t) \times 2^{e_{min}}$, where $e_{min} = 1 - bias$.

`NaN`s are used to represent the result of an invalid operation (such as $\infty - \infty$) and are described by $e = 2^w - 1$ and a non-zero t. There are two types of `NaN`s: quiet `NaN`s (`qNaN`s) and signalling `NaN`s (`sNaN`s). The significand encodes the type of `NaN` and debug information in some implementations. Operations generally propagate `qNaN`s and quiet `sNaN`s. If one of the operands is `qNaN`, the result is `qNaN`. If the operand is an `sNaN`, an exception may be raised, or that operand may be converted to a `qNaN`.

Floating-point exceptions occur in situations such as division by zero or computation involving an `sNaN`. By default, floating-point exceptions do not alter control-flow but raise a status flag and return a default result (e.g. a `qNaN`).

Floating-point instructions In its assembly language, LLVM defines several instructions for binary floating-point operations (`fadd`, `fsub`, `fmul`, `fdiv`, ...), conversion instructions (`fptrunc`, `fpext`, `fptoui`, `uitofp`, ...), and allows floating-point arguments in other operations (e.g. `select`). We assert that floating-point instructions cannot generate poison values (values that cause undefined behavior for instructions that depend on them) or result in undefined behavior. The documentation is not entirely clear but our interpretation is that undefined behavior does not occur in the absence of `sNaN`s and that `sNaN`s are not fully supported.

While IEEE 754-2008 defines different rounding modes, LLVM does not yet allow users to specify them. As a consequence, the rounding performed by `fptrunc` (casting a floating-point value to a smaller floating-point type) is undefined for inexact results.

Flag	Description	Formula
`nnan`	Assume arguments and result are not `NaN`. Result undefined over `NaN`s.	`ite (or (isNaN a) (isNaN b) (isNaN r) (x (_ FP <ebits> <sbits>)) r`
`ninf`	Assume arguments and result are not $\pm\infty$. Result undefined over $\pm\infty$.	`ite (or (isInf a) (isInf b) (isInf r)) (x (_ FP <ebits> <sbits>)) r`
`nsz`	Allow optimizations to treat the sign of a zero argument or result as insignificant.	`or (a = b) (and (isZero a) (isZero b))`

Table 1. Fast-math flags that LifeJacket supports. The `isNaN` and `isInf` are not part of the SMT-LIB standard but are supported in Z3's Python interface and used for illustration purposes here. The variable `x` is a fresh, unconstrained variable, `a` and `b` are the SMT formulas of the operands, `r` of the result. The formula for `nsz` replaces the standard equality check `a = b`.

Fast-math flags Some programs either do not depend on the exact semantics of special floating-point values or do not expect special values (such as `NaN`) to occur. To specify these cases, LLVM binary operators can be annotated with *fast-math flags*, which allow LLVM to do additional optimizations with the knowledge that special values will not occur. Table 1 summarizes the fast-math flags that LifeJacket supports. There are two additional flags, `arcp` (allows replacing arguments of a division with the reciprocal) and `fast` (allows imprecise optimizations), that we do not support.

Discussion The properties of floating-point arithmetic discussed in this section hint at how difficult it is to manually reason about floating-point optimizations. The floating-point standard is complex, so compilers do not always follow it completely—as we mentioned earlier, LLVM does not currently support different rounding modes.[2] Similarly, it does not yet support access to the floating-point environment, which makes reliable checks for floating-point exceptions in `clang` impossible, for example. This runs counter to the IEEE standard, which defines reproducability as including "invalid operation," "division by zero," and "overflow" exceptions.

3.2 Verifying Transformations with Alive

Alive is a tool that verifies peephole optimizations on LLVM's intermediate representation; these optimizations are expressed (as input) in a domain-specific language. At a high level, verifying an optimization with Alive takes the following steps:

1. The user specifies a new or an existing LLVM optimization using the Alive language.
2. Alive translates the optimization into a series of SMT queries that express the equivalence of the source and the target.
3. Alive uses Z3, an SMT solver, to check whether any combination of inputs makes the source and target disagree. If the optimization is incorrect, Alive returns a counter-example that breaks the optimization.

Alive specializes in peephole optimizations that are highly local and do not alter the control-flow graph of a program. This type of optimization is performed by the LLVM InstCombine pass in `lib/Transforms/InstCombine` and InstructionSimplify in `lib/Analysis`.

Alive can also generate code for an optimizer pass that performs all of the verified optimizations. We do not discuss this feature further since LifeJacket does not yet support it for floating-point optimizations. In the following, we discuss the Alive language and the role of SMT solvers in proving optimization correctness.

Specifying transformations with the Alive language In Alive, each transformation consists of a list of preconditions, a source template, and a target template. Alive verifies whether it is safe to replace the instructions in the source with the instructions in the target given that the preconditions hold. Figure 1 is an example of a transformation written in Alive. It has no preconditions, so it always applies. Its source and target instructions are delimited by "`=>`".

Preconditions are logical expressions enforced by the compiler at compile-time and Alive takes them for granted. The predicate `isNormal(%x)` in the precondition, for example, means that the optimization only applies when `%x` is a normal floating-point value.

In Alive, the arguments for instructions are either inputs (e.g. `%x`), constant expressions (e.g. `C` or `C1 + C2`), or immediate values (e.g. `0.0`). Inputs model LLVM registers, constant expressions correspond to computations that LLVM performs at compile-time, and immediate values are constants in the LLVM source code and thus known to Alive at verification-time. Inputs and constant expressions may be used as subjects for predicates in the precondition. Alive interprets the instructions in the sources and targets as expression trees, so the order of instructions does not matter, only the dependencies. Verifying the equivalence of the source and the target is done on the root of the trees.

In contrast to actual LLVM code, the Alive language does not require type information for instructions and inputs. Instead, it uses the types expected by instructions to restrict types and bit-widths of types. Then, it issues an SMT query that encodes these constraints to infer all possible types and sizes of registers, constants, and values. This mirrors the fact that LLVM optimizations often apply to multiple bit-widths and makes specifying optimizations less repetitive. Alive instantiates the source and target templates with the possible type and size combinations and verifies each instance.

Undefined values (`undef`) in LLVM represent values that are not defined, such as results of reads from uninitialized memory. They return arbitrary bit-patterns when read and may be of any type. For each `undef` value in the target template, Alive has to verify that any value can be produced, and for each `undef` value in the source, Alive may assume any convenient value. Figure 2 is a known incorrect optimization in LLVM that LifeJacket confirms and that illustrates this concept: The source template cannot produce all possible bit-patterns, so it cannot be replaced with `undef`.[3]

Incorrect:	Correct:

`%r = fdiv %x, undef`
`=>`
`%r = undef`
	`%r = fdiv %x, undef`
	`=>`
	`%r = NaN`

Figure 2. Example of a problematic optimization using `undef` on the left and a better version on the right. If `%x` is `NaN` then `%r` can only be `NaN`, so `%r` cannot be `undef`.

Verifying transformations with SMT solvers Alive translates the source and target template into SMT formulas. For each possible combination of variable types in the templates, it creates SMT formulas for definedness constraints, poison-free constraints, and the

[2] More details: `http://lists.llvm.org/pipermail/llvm-dev/2016-February/094869.html`.

[3] Discussion on this optimization: `https://groups.google.com/d/topic/llvm-dev/iRb0gxroT9o/discussion`

return values for the source and target. Alive checks the definedness and poison-free constraints of the source and target for consistency. These checks are not directly relevant to floating-point arithmetic, so we do not discuss them further. Instead, we deal more directly with the execution values of the source and target.

An optimization is only correct if the source and the target always produce the same value. To check this property, Alive asks an SMT solver to verify that `preconditions` $\wedge$ `src_formula` $\neq$ `tgt_formula` is unsatisfiable—that there is no assignment that makes the formula true. If there is, the optimization is *incorrect*: there is an assignment for which the source value is different from the target value. When Alive encounters an incorrect optimization, it uses the output of the SMT solver to return a counterexample. This counterexample consists of input and constant assignments that lead to different source and target values.

Ultimately, Alive relies on the Z3 SMT solver to determine whether an optimization is correct. LifeJacket would have been impossible without Z3's floating-point support, which was added less than a year ago in version 4.4.0 and implements the SMT-LIB standard for floating-point arithmetic [12].

4. Implementation

Our implementation extends Alive in four major ways: it adds support for floating-point types, floating-point instructions, floating-point predicates, and fast-math flags. In the following, we describe our work in those areas, briefly comment on our experience with floating-point support in Z3, and conclude with a discussion of the limitations of the current version of LifeJacket.

Floating-point types LifeJacket implements support for `half`, `single`, and `double` floating-point types. Alive itself provides support for integer and pointer types of arbitrary bit-widths up to 64 bit. Following the philosophy of the original implementation, we do not require users to explicitly annotate floating-point types. Instead, we use a logical disjunction in the SMT formula for type constraints to limit floating-point types to bit-widths of 16, 32, or 64 bits. Then, we use Alive's existing mechanisms to determine all possible type combinations for each optimization (as discussed in Section 3.2).

Adding a new type requires us to relax some assumptions, e.g. that the arguments of `select` are integers. Additionally, we modify the parser to support floating-point immediate values.

Floating-point predicates and constant functions LifeJacket adds precondition predicates and constant functions related to floating-point arithmetic.

Recall that preconditions are logical formulas that describe facts that must be true for an optimization to apply; they are fulfilled by LLVM and assumed by Alive. In the context of floating-point optimizations, preconditions may include predicates about the type of a floating-point value (e.g. `isNormal(%x)` to make sure that `%x` is a normal floating-point number) or checks to ensure that conversions are lossless.

Constant functions mirror computation performed by LLVM at compile-time and are evaluated by Alive symbolically at verification-time. For example, the constant function `fptosi(C)` (not to be confused with the instruction) converts a floating-point number to a signed integer, corresponding to a conversion that LLVM does at compile time. Constant expressions (expressions that contain constant functions) can be assigned to registers in the target template, mirroring the common pattern of optimizing operations by partially evaluating them at compile-time.

In contrast to Alive, LifeJacket supports precondition predicates that refer to constant expressions in target templates. For example, some optimizations have restrictions about precise conversions, and we express those restrictions in the precondition. If the target converts a floating-point constant to an integer with `%c = fptosi(C)`, then the precondition can ensure that the conversion is lossless by including `sitofp(%c) == C` (which guarantees that converting the number back and forth results in the original number). If the precondition were `sitofp(fptosi(C)) == C`, then `fptosi(C)` in the precondition and in `%c` would be independent. Because of their independence, `%c` would stay unrestricted and LifeJacket would find a counterexample where the bit-width of `%c` was too small to represent the converted value.

```
double fmod(double x, double y) {
  double result;
  result = remainder(fabs(x), (y = fabs(y)));
  if (signbit(result)) result += y;
  return copysign(result, x);
}
```

```
(= abs_y (abs y))
(= r (remainder (abs x) abs_y))
(= r' (ite (isNeg r) (+ RNE r abs_y) r))
(= fmod (ite (xor (isNeg x) (isNeg r')) (- r') r))
```

Figure 3. The `fmod` function implemented using IEEE `remainder` as suggested by the C standard and an informal representation of the implementation used by LifeJacket.

Floating-point instructions Our implementation supports binary floating-point instructions (`fadd`, `fsub`, `fmul`, `fdiv`, and `frem`), conversions involving floating-point numbers (`fptrunc`, `fpext`, `fptoui`, `fptosi`, `uitofp`, `sitofp`), the `fabs` intrinsic, and floating-point comparisons (`fcmp`). Most of these instructions directly correspond to operations that the SMT-LIB for floating-point standard supports, so translating them to SMT formulas is straightforward. Next, we discuss our support for `frem`, `fcmp`, conversions, and the equivalence check for floating-point optimizations.

The `frem` instruction does not correspond to `remainder` as defined by IEEE 754 but rather to `fmod` in the C POSIX library, so translating it to an SMT formula involves multiple operations. Both `fmod` and `remainder` calculate $x - n * y$ (where n is $\frac{x}{y}$), but `fmod` rounds toward zero whereas `remainder` rounds to the nearest value and ties to even. Figure 3 shows how the C standard defines `fmod` in terms of `remainder` for `doubles` [9, §F.10.7.1] and the corresponding SMT formula that LifeJacket implements. The formula uses a fixed rounding mode because the rounding mode of the environment does not affect `fmod`.

The `fcmp` instruction compares two floating-point values. In addition to the two floating-point values, it expects a third operand, the *condition code*. The condition code determines the type of comparison. There are two genres of comparison: *ordered* comparisons can only be true if none of the inputs are `NaN` and *unordered* comparisons are true if any of the inputs is `NaN`. LLVM supports an ordered version and an unordered version of the usual comparisons such as equality, inequality, greater-than, etc. Additionally, there are condition codes that just check whether both inputs are not `NaN` (`ord`) or any of the inputs are `NaN` (`uno`).

Optimizations involving comparisons often apply to multiple condition codes. To allow users to efficiently describe such optimizations, LifeJacket supports predicates in the precondition that describe the applicable set of condition codes. For example, there are predicates for constraining the set of condition codes to either ordered or unordered conditions (`ordered(CC)`/`unordered(CC)`). We also support predicates that express relationships between multiple condition codes. This is useful, for example, to describe an optimization that performs a multiplication by negative one on both sides: to replace a comparison between `-x` and `C` with condition

code C1 by a comparison between x and -C with the condition code C2, we use the swap(C1, C2) predicate.

When no sensible conversion between floating-point values and integers is possible, LLVM defaults to returning undef. For conversions from floating-point to integer value (signed or unsigned), LifeJacket checks whether the (symbolic) floating-point value is NaN, $\pm\infty$, too small, or too large, and returns undef if necessary. Conversions from integer to floating-point values similarly return undef for values that are too small or too large.

Recall that LifeJacket must determine the unsatisfiability of precondition $\wedge$ src_formula $\neq$ tgt_formula to verify optimizations. The SMT-LIB standard defines two equality operators for floating-point, one implementing bit-wise equality, and one implementing the IEEE equality operator. The latter operator treats signed zeros as equal and NaNs as different, so using it to verify optimizations would not work, since it would accept optimizations that produce different zeros and reject source-target pairs that both produce NaN. The bit-wise equality works, because SMT-LIB uses a single NaN value (recall that there are multiple bit-patterns that correspond to NaN). While this is convenient, it also means that we cannot model different NaNs. We discuss the implications of this limitation later.

Fast-math flags LifeJacket currently supports three of the five fast-math flags that LLVM implements: nnan, ninf, and nsz.

LifeJacket handles the nnan and ninf flags in a similar way by modifying the SMT formula for the instruction on which the flag appears. As Table 1 shows, if the instruction's arguments or result is a NaN or $\pm\infty$, respectively, the formula returns a fresh unconstrained variable that it treats as an undef value. This is a direct translation from the description in the language reference and works for root and non-root instructions.

The nsz flag is different: instead of relaxing the requirements for the behavior of the instruction for certain inputs and results, it states that the sign of a zero value can be ignored. This primarily affects how LifeJacket compares the source and target values: it adds a logical conjunction to the SMT query that states that the source and target values are only different if both are nonzero (shown in Table 1). The flag itself has no effect on zero values at runtime, meaning that it does not affect the computation performed by instructions with the flag. Thus, we do not change the SMT formula for the instruction.

Since the nsz flag has no direct effect on how LLVM does matching, this flag *also* does not change the significance of the sign of immediate zeros (e.g. +0.0) in the optimization templates. Instead, we mirror how LLVM determines whether an optimization applies. In LLVM, optimizations that match a certain sign of zero do not automatically apply to other zeros when the nsz flag is set. For example, an optimization that applies to fadd x, -0.0 does *not* automatically apply to fadd nsz x, +0.0. If applicable, developers explicitly match any zero if the nsz flag is set. We mirror this design by implementing an AnyZero(C) predicate, which makes C negative or positive zero.

Limitations While Section 5 shows that LifeJacket is a useful tool, it does not support all floating-point types and imprecise optimizations, uses a fixed rounding mode, and does not model floating-point exceptions and debug information in NaNs.

Currently, LifeJacket does not support LLVM's vectors and the two 128-bit and the 80-bit floating-point types. Supporting those would likely not require fundamental changes.

There are many imprecise optimizations in LLVM. These optimizations need a different style of verification because they do not make any guarantees about how much they affect the program output. A possible way to deal with these optimizations would be to verify their correctness for real numbers and estimate accuracy changes by randomly sampling inputs, similarly to Herbie's approach [11].

LifeJacket's verification ultimately relies on the SMT-LIB standard for floating-point arithmetic. The standard corresponds to IEEE 754-2008 but it only defines a single NaN value and does not distinguish between signalling and quiet NaNs. Thus, our implementation cannot verify whether an operation with NaN operands returns one of the input NaNs, correctly propagating debug information encoded in the NaN, as recommended by the IEEE standard. In practice, LLVM does not attempt to preserve information in NaNs, so this limitation does not affect our ability to verify LLVM optimizations. We do not model floating-point exceptions, either, since LLVM does not currently make guarantees about handling floating-point exceptions. Floating-point exceptions could be verified with separate SMT queries, similar to how Alive verifies definedness.

LifeJacket currently rounds to nearest and ties to the nearest even digit, mirroring the most common rounding mode. Even though LLVM does not yet support different rounding modes, we are planning to add this support to LifeJacket soon.

Limited type and rounding mode support and missing floating-point exceptions make our implementation unsound at worst: LifeJacket may label some incorrect optimizations as correct, but optimizations labelled as incorrect are certainly wrong.

Working with Z3 We found Z3's floating-point support very effective despite its relative youth. We use the Z3 master branch because of issues in the most recent release's implementation and Python API. During the development of LifeJacket, we reported issues—all of which were fixed quickly—and fixed a few issues in the Python API ourselves. This suggests that LifeJacket is an interesting test case for floating-point support in SMT solvers.

5. Evaluation

To evaluate LifeJacket, we translated 54 optimizations from LLVM 3.7.1 into the Alive language and tried to verify them. We discovered 8 incorrect optimizations and verified 43 optimizations to be correct. In the following, we outline the optimizations that we checked and describe the bugs that we found.

We performed our evaluation on a machine with an Intel i3-4160 CPU and 8 GB RAM, running Ubuntu 15.10. We compiled Z3 commit 2250728 with GCC 5.2.1, the default compiler, used the qffp tactic for quantifier-free formulas and bv for formulas with quantifiers, and chose a 30 minute timeout for individual SMT queries. Table 2 summarizes the results for different source files: AddSub contains optimizations with fadd/fsub at the root, MulDivRem with fmul/fdiv/frem, Compares deals with fcmps and Simplify contains simple optimizations for all instructions.

Using this process, LifeJacket found 43 out of 54 optimizations to be correct. LifeJacket timed out on three optimizations. The AddSub optimization that times out contains a sitofp instruction and verification is slow for integers with a large bit-width. The two MulDivRem optimizations that timeout both contain nsz flags and AnyZero predicates. Similar optimizations without those features do not timeout. Out of the eight optimizations that LifeJacket found to be incorrect, five had been previously reported. The bug in Figure 1 had already been fixed in a newer version of LLVM when we discovered it. The rest of the reported bugs resemble the example in Figure 2 and are all caused by an unjustified undef in the target (PR26862/PR26863). Figure 4 depicts the three previously unreported incorrect optimizations that we reported to the LLVM developers. We discuss these bugs in the next paragraphs.

PR26958 optimizes $(0 - x) + x$ to 0. The implementation of this optimization requires that the nnan and the ninf flag each appear at least once on the source instructions. We translate four variants of this instruction: one where both flags are on fsub, one where both are on fadd and two where each instruction has one of the flags. As it turns out, it is not enough to have both flags on either

```
Name: PR26958
Precondition: AnyZero(C0)
%a = fsub nnan ninf C0, %x
%r = fadd %x, %a
  =>
%r = 0.0
```

```
Name: PR26943
%a = select i1 %c, 0.0, C
%r = frem %x, %a
  =>
%r = frem %x, C
```

```
Name: PR27036
Precondition: hasOneUse(%a) &&
  hasOneUse(%b) &&
  WillNotOverflowSignedAdd(%x, %y)
%a = sitofp %x
%b = sitofp %y
%r = fadd %a, %b
  =>
%c = add nsw %x, %y
%r = sitofp %c
```

Figure 4. New bugs in LLVM 3.7.1 found by LifeJacket.

File	Verified	Timeouts	Bugs
AddSub	7	1	1
MulDivRem	3	2	1
Compares	11	0	0
Simplify	22	0	6
Total	43	3	8

Table 2. Number of optimizations verified, timeouts, and bugs.

of the instructions. For the case where both flags are on `fsub`, the transformation is invalid if `%x` is `NaN` or $\pm\infty$. The `nnan` and `ninf` flags require the optimized program to retain defined behavior over `NaN` and $\pm\infty$, so `%r` must be `0.0` even for those inputs (if they resulted in undefined behavior, any result would be correct). If `%x` is `NaN`, however, then there is no value for `%a` that would result in `%r` being `0.0` because `NaN` added to any other number is `NaN`.

`PR26958` optimizes `fmod(x, c ? 0 : C)` to `fmod(x, C)` (`select` acts like a ternary and `frem` corresponds to `fmod`). The implementation of this optimization shares its code with the same optimization for the `rem` instruction that deals with integers. For integers, `rem %x, 0` results in undefined behavior, so the optimization is valid. The POSIX standard specifies that `fmod(x, 0.0)` returns `NaN`, though, so the optimization is incorrect for `frem` because `%r` must be `NaN` and not `frem %x, C` if `%a` is `0.0`.

`PR27036` illustrates the last incorrect optimization that LifeJacket identified. It transforms `(float) x + (float) y` into `(float) (x + y)`, replacing an `fadd` instruction with a more efficient `add`. This transformation is invalid, though, since adding two rounded numbers is not equivalent to adding two numbers and rounding the result. For example, assuming 16-bit floating-point numbers, let `%x = -4095` and `%y = 17`. In the portion of the source formula `%a = sitofp %a`, `%a` cannot store an exact number and stores `-4094` instead. The target formula, though, can accurately represent the result `-4078` of the addition.

Our results confirm that it is difficult to write correct floating-point optimizations; we found bugs in almost all the LLVM files from which we collected our optimizations. Unsurprisingly, all of these bugs relate to floating-point specific properties such as rounding, `NaN`, $\pm\infty$ inputs, and signed zeros. This demonstrates that these edge cases are difficult to reason about.

6. Conclusion

In an ideal world, programming languages and compilers are boring. They do what the user expects. They exhibit the same behavior with and without optimization, at all optimization levels, and on all hardware. "Boring," however, is surprisingly difficult to achieve, especially in the context of the complicated semantics of floating-point arithmetic. With LifeJacket, we hope to make LLVM's precise floating-point optimizations more predictable (and boring) by automatically checking them for correctness.

Acknowledgments

We thank Christopher Aberger, Dawson Engler, John Regehr, Nelson Beebe, Nuno Lopes, Pascal Cuoq, Xi Wang, and Zvonimir Rakamaric. This research is supported by 1165929-1-DJYBJ and DARPA Subcontract No. 1190029-276707.

References

[1] Souper. `https://github.com/google/souper`.

[2] LLVM language reference manual. `http://llvm.org/releases/3.7.1/docs/LangRef.html`, 2016.

[3] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 2015.

[4] S. Buchwald. Optgen: A generator for local optimizations. In *24th Internation Conference on Compiler Construction (CC)*, 2015.

[5] E. Darulova and V. Kuncak. Sound compilation of reals. In *Acm Sigplan Notices*, volume 49, pages 235–248. ACM, 2014.

[6] S. N. David Menendez and A. Gupta. Alive-FP: Automated verification of floating point based peephole optimizations in LLVM. Technical report, 2016.

[7] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*. 2008.

[8] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *CSUR*, 1991.

[9] I. JTC1/SC22/WG14. ISO/IEC 9899:2011, Programming languages - C. Technical report, 2011.

[10] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. In *PLDI*, 2015.

[11] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI*, 2015.

[12] P. Rümmer and T. Wahl. An SMT-LIB theory of binary floating-point arithmetic. In *SMT Workshop*, 2010.

[13] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices*, 49(6), 2014.

[14] C. Severance. An interview with the old man of floating point. *IEEE Computer*, pages 114–115, 1998.

[15] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM*, pages 532–550. 2015.

[16] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *SOSP*, 2013.

[17] A. A. Wonyeol Lee, Rahul Sharma. Verifying bit manipulations of floating-point. In *PLDI*, 2016.

[18] A. Zeljić, C. M. Wintersteiger, and P. Rümmer. Approximations for model construction. In *Automated Reasoning*, pages 344–359. 2014.