Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM

David Menendez[1], Santosh Nagarakatte[1], and Aarti Gupta[2]

[1] Rutgers University, New Brunswick, USA,
{davemm, santosh.nagarakatte}@cs.rutgers.edu
[2] Princeton University, Princeton, USA,
aartig@cs.princeton.edu

Abstract. Peephole optimizations optimize and canonicalize code to enable other optimizations but are error-prone. Our prior research on Alive, a domain-specific language for specifying LLVM's peephole optimizations, automatically verifies the correctness of integer-based peephole optimizations and generates C++ code for use within LLVM. This paper proposes Alive-FP, an automated verification framework for floating point based peephole optimizations in LLVM. Alive-FP handles a class of floating point optimizations and fast-math optimizations involving signed zeros, not-a-number, and infinities, which do not result in loss of accuracy. This paper provides multiple encodings for various floating point operations to account for the various kinds of undefined behavior and under-specification in the LLVM's language reference manual. We have translated all optimizations that belong to this category into Alive-FP. In this process, we have discovered seven wrong optimizations in LLVM.

1 Introduction

Compilers perform numerous optimizations transforming programs through multiple intermediate representations to produce efficient code. Compilers are intended to preserve the semantics of source programs through such transformations and optimizations. However, modern compilers are error-prone similar to other large software systems. Recent random testing approaches [1, 2] have found numerous bugs in mainstream compilers. These bugs range from compiler crashes to silent generation of incorrect programs. Among them, peephole optimizations are persistent source of compiler bugs [1, 2].

Peephole optimizations perform local algebraic simplifications of code, clean up code produced by other stages, and canonicalize code to enable further optimizations. Any misunderstanding in the semantics of the instructions, overlooked corner cases, and the interaction of algebraic simplification with undefined behavior results in compiler bugs. Furthermore, modern compilers have numerous such peephole optimizations (*e. g.*, the LLVM compiler has more than a thousand such peephole optimizations performed by the `InstCombine` and the `InstSimplify` passes).

Our prior research on the Alive domain-specific language addresses a part of this problem of developing correct peephole optimizations in LLVM [3]. An Alive optimization is of the form *source* $\implies$ *target* with an optional precondition. The optimization checks the input code for a directed acyclic graph (DAG) of the form *source* and replaces it with the DAG specified by the *target* (see Section 2.1 for more details). Optimizations expressed in Alive are verified by encoding them as first-order logic formulae whose validity is checked using Satisfiability Modulo Theories (SMT) solvers. Further, the Alive interpreter generates C++ code for use within the LLVM compiler to provide competitive compilation time.

In our prior work, we restricted Alive to integer optimizations because floating point (FP) support with SMT solvers was either non-existent or not mature at that point in time. This paper proposes Alive-FP, an extension of Alive, that adds automated verification for a class of FP optimizations in LLVM. Alive-FP leverages the Floating Point Arithmetic (FPA) [4] theory in SMT-LIB, which has been supported by SMT solvers like Z3 [5], to perform automated reasoning.

A floating point number approximates a real number. The IEEE-754 standard [6] provides portability of FP operations across various hardware and software components by standardizing the representation and operations. Today, most processors support FP representation natively in hardware. To represent large and small values, floating point numbers in the IEEE-754 are represented in the form: $(-1)^s \times M \times 2^E$, where s determines the sign, significand M is a fractional binary number (ranges between $[1, 2)$ or $[0, 1)$), and E is the exponent that weights the value (see Section 2.2 for a background on FP numbers). The representation has normal values (the common case), denormal values (values close to zero), two zeros ($+0.0$ and -0.0), and special values such as positive infinity, negative infinity, and not-a-number (NaN). Further, the standard has specific rules on ordering and equality of numbers, and generation and propagation of special values. Most algebraic identities for real numbers are not accurate with FP numbers due to rounding errors (addition is associative with reals but is not with FP). Unfortunately, most programmers are not comfortable with these arcane details (see the seminal paper on FP [7] for more information).

Peephole optimizations in LLVM related to FP can be classified into two categories: optimizations that preserve bitwise precision (*i. e.*, there is no loss in accuracy with the optimization) and optimizations that do not preserve bit precision (*i. e.*, sacrifice some accuracy). The optimizations that preserve bit precision can further be classified into two categories: optimizations that are identities according to the IEEE-754 standard and optimizations that preserve bit precision in the absence of special values. LLVM, by default, enables optimizations that are identities according to the standard. LLVM enables bit-precise optimizations in the absence of special values with explicit fast-math attributes `nnan` (assume no NaNs), `nsz` (assume no distinction between two zeros), and `ninf` (assume no infinities). LLVM enables optimizations that are not bit-precise with fast-math attributes `arcp` (allow reciprocals) and `fast`.

In this paper, we are focused on verifying the correctness of bit-precise floating point based peephole optimizations in LLVM. We propose Alive-FP, a new domain-specific language, that extends Alive with FP instructions, types, and predicates (see Section 3) and allows the use of both integer and FP operations. Alive-FP proves the correctness of FP optimizations by encoding them into first-order logic formulae in the SMT-LIB theory of floating point arithmetic (FPA), whose validity is checked using Z3 [5]. We highlight the under-specification in the LLVM's language reference manual about undefined behavior and rounding modes with floating point operations. Given this under-specification about undefined behavior and rounding modes, we provide multiple encodings for the FP instructions and attributes (see Section 4) into FPA theory. We have translated 104 bit-precise FP peephole optimizations from LLVM into Alive-FP (see Section 5). In this process, Alive-FP has discovered seven previously unknown floating point bugs [8–12]. We have added new features to Alive-FP on request from developers (*e. g.*, `bitcast` with FP). Alive-FP is open source [13].

2 Background

This section provides background on Alive [3], which we build upon in this work. We also provide a quick primer on the FP representation in the IEEE standard.

2.1 Alive Domain-Specific Language

Alive [3] is a domain-specific language for specifying LLVM's peephole optimizations. The Alive interpreter automatically checks the correctness of an optimization by encoding it into a formula in first-order logic whose validity is checked using an SMT solver. It also generates C++ code for use within LLVM, which implements the optimization.

Syntax. Alive syntax is similar to the LLVM intermediate representation (IR), because LLVM developers are already familiar with it. Alive optimizations are specified as `source` ⇒ `target`, with an optional precondition. An Alive optimization is a directed, acyclic graph (DAG), with two *root* nodes, which have only outgoing edges. One root is the source, which is the instruction to be replaced, and the other is the target, which replaces it. To indicate that the source root replaces the target, they must have the same name. The leaves in the DAG are input variables, symbolic constants, or constant expressions. The target may refer to instructions defined in the source and create new instructions, which will be added when the source root is replaced. An example Alive optimization is shown in Figure 1. The root `%d` in the source is replaced with the root in the target when the precondition is satisfied (*i. e.*, `isSignBit(C1)`, where `C1` is a symbolic constant). Alive preconditions consist of built-in predicates, equalities, signed/unsigned inequalities, and predicates

```
Pre: isSignBit(C1)
%b = xor %a, C1
%d = add %b, C2
=⇒
%d = add %a, C1 ˆ C2
```

Fig. 1. An Alive optimization. The predicate `isSignBit(C1)` is true when C1 is all zeros except for the sign bit.

representing the result of dataflow analyses. An Alive optimization is parametric over types and bit widths. Hence, the Alive interpreter checks the correctness of the optimization for feasible types and bit widths.

Undefined behavior and compiler optimizations. Most compiler bugs result from a misunderstanding in the semantics, especially regarding various kinds of undefined behavior [3]. The semantics of an instruction specifies when it is well-defined. LLVM optimizes the program with the assumption that the programmer never intends to have undefined behavior in the program [14–16].

To aggressively optimize well-defined programs, LLVM IR has undefined behavior with catch-fire semantics, the `undef` value, and poison values. Undefined behavior in LLVM is similar to undefined behavior in C/C++, where program can perform any action in the presence of undefined behavior. Typically, a program with undefined behavior will cause a fault/trap (*e. g.*, division by zero) at runtime.

The `undef` value indicates that the program reads an uninitialized memory location and the program is expected to be well-defined for any value chosen for the `undef` value. The `undef` value is equivalent to reading a hardware register at an arbitrary instant of time.

A *poison* value represents the fact that an instruction, which does not have side effects, has produced a condition that results in undefined behavior. Unlike `undef` values, there is no explicit way of representing a poison value in the IR. A poison value indicates ephemeral effects of certain incorrect operations (*e. g.*, signed overflow with an instruction expected not to have signed overflows). LLVM has instruction attributes such as `nsw` (no signed wrap), `nuw` (no unsigned wrap), and `exact`, which produce poison values on incorrect operations. An arithmetic instruction with the no signed wrap attribute produces a poison value on signed overflows [3]. Unlike `undef` values, poison propagates with dependencies. A poison value triggers undefined behavior with catch-fire semantics when it produces an externally-visible effect (*e. g.*, atomic loads/stores). Hence, poison values that do not affect a program's externally visible behavior are allowed in a well-defined program.

From a Alive perspective, when the source contains a `undef` value, the compiler can pick a value for the `undef` in the source that makes the optimization valid. In contrast, when the source produces a poison value, the compiler can replace the target with anything as the source is not well-defined.

Correctness of an Alive optimization. Alive's verification engine reasons about the correctness of optimizations taking into account various undefinedness conditions, which eases the job of the compiler writer. Given an optimization, the Alive interpreter instantiates candidate types for the optimization. The Alive interpreter encodes the Alive optimization with concrete types into first order logic formulae. The validity of the formulae imply the correctness of the optimization. The interpreter generates the following validity checks when the source template is well-defined, poison-free, and the precondition is satisfied: (1) the target is well-defined, (2) the target is poison-free, and (3) the source and target root of the DAG produce the same value, and (4) the memory states in the source and the target are equivalent (for optimizations involving memory operations). These checks are performed for each feasible type instantiation.

2.2 Floating Point Representation and Arithmetic

A floating point number is an approximation of a real number. Although many computer manufacturers had their own conventions for floating point earlier, IEEE-754 [6] has become the standard for floating point representation. The IEEE standard represents a floating point number in the form: $(-1)^s \times M \times 2^E$, where s determines the sign of the number, significand M is a fractional binary number ranging either between $[1, 2)$ or $[0, 1)$, and exponent E weights the value with a power of 2 (can be negative). The bit representation uses a single bit for sign s, k bits for the encoding the exponent E and n-bit fractional field for encoding the significand M (*e.g.*, 32-bit float has 1 sign bit, $k = 8$, and n= 23). The value encoded by the bit representation is divided into three categories depending on the exponent field.

Normalized values. When the bit pattern for the exponent is neither all zeros nor all ones, the floating number is a normalized value. The exponent field is interpreted as a signed integer in biased form (*i. e.*, $E = e - bias$), where e is the unsigned number from the bit pattern in the exponent field. The $bias$ is $2^{k-1} - 1$. The significand M is $1.f$, where f is the fractional field in the representation. The normalized value is $(-1)^s \times 1.f \times 2^{(e-bias)}$.

Denormalized values. When the bit pattern in the exponent field is all zeros, then the number is a denormal value. In this case the exponent is $1 - bias$. The significand value is f, the fractional part without the leading 1. Denormal values are used to represent zero and values close to zero. When all the exponent bits and fractional bits in the bit pattern are zero, then it represents a zero. There is a positive zero and negative zero depending on value of the sign bit.

Special values. When the bit pattern in the exponent field is all ones, then it represents special values. There are three special values: positive-infinity, negative-infinity, and not-a-number (NaN). When the fractional field is all zeros, then the number is a positive-infinity or a negative infinity depending on the sign. When the fractional part is not all zeros, then it is a NaN. The standard provides specific rules for equality, generation, and propagation of these special values.

Rounding modes. The floating point number has limited range and precision compared to real numbers. Rounding modes specify a systematic method for finding the "closest" matching value that can be represented in a floating point format. The IEEE-754 standard specifies the following rounding modes: round toward nearest with ties going to the even value (RNE), round towards nearest with ties going away from zero (RNA), round towards positive (RTP), round towards negative (RTN), and round toward zero (RTZ). The rounding mode is described with respect to a program block in the standard. The FPA theory in SMT-LIB also includes these rounding modes, and parameterizes most operations with a rounding mode. In contrast, the LLVM language reference does not explicitly state the rounding mode for its FP operations.

$prog ::= pre\ nl\ stmt \implies stmt$
$stmt ::= stmt\ nl\ stmt \mid reg = inst \mid reg = op$
$inst ::= binop\ \overline{attr}\ op, op \mid fpbop\ \overline{fmf}\ op, op \mid$
$\quad conv\ op \mid \mathsf{select}\ op, op, op \mid$
$\quad \mathsf{icmp}\ cond\ op, op \mid \mathsf{fcmp}\ \overline{fmf}\ fcnd\ op, op$
$typ ::= \mathsf{i}sz \mid \mathsf{half} \mid \mathsf{float} \mid \mathsf{double} \mid \mathsf{fp128} \mid$
$\quad \mathsf{x86\_fp80}$
$binop ::= \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{udiv} \mid \mathsf{sdiv} \mid$
$\quad \mathsf{urem} \mid \mathsf{srem} \mid \mathsf{shl} \mid \mathsf{lshr} \mid \mathsf{ashr} \mid$
$\quad \mathsf{and} \mid \mathsf{or} \mid \mathsf{xor}$
$attr ::= \mathsf{nsw} \mid \mathsf{nuw} \mid \mathsf{exact}$

$fpbop ::= \mathsf{fadd} \mid \mathsf{fsub} \mid \mathsf{fmul} \mid \mathsf{fdiv} \mid \mathsf{frem}$
$fmf ::= \mathsf{nnan} \mid \mathsf{ninf} \mid \mathsf{nsz} \mid \mathsf{arcp} \mid \mathsf{fast}$
$op ::= reg \mid constant \mid \mathsf{undef} \mid$
$\quad \mathsf{nan} \mid \mathsf{inf} \mid -\mathsf{inf}$
$conv ::= \mathsf{zext} \mid \mathsf{sext} \mid \mathsf{trunc} \mid \mathsf{fpext} \mid \mathsf{fptrunc} \mid$
$\quad \mathsf{fptosi} \mid \mathsf{fptoui} \mid \mathsf{sitofp} \mid \mathsf{uitofp} \mid$
$\quad \mathsf{bitcast} \mid \mathsf{inttoptr} \mid \mathsf{ptrtoint}$
$cond ::= \mathsf{eq} \mid \mathsf{ne} \mid \mathsf{ugt} \mid \mathsf{uge} \mid \mathsf{ult} \mid$
$\quad \mathsf{ule} \mid \mathsf{sgt} \mid \mathsf{sge} \mid \mathsf{slt} \mid \mathsf{sle}$
$fcnd ::= \mathsf{oeq} \mid \mathsf{one} \mid \mathsf{ogt} \mid \mathsf{oge} \mid \mathsf{olt} \mid \mathsf{ole} \mid \mathsf{ord} \mid$
$\quad \mathsf{ueq} \mid \mathsf{une} \mid \mathsf{ugt} \mid \mathsf{uge} \mid \mathsf{ult} \mid \mathsf{ule} \mid \mathsf{uno}$

Fig. 2. Partial Alive-FP syntax. The new additions in Alive-FP when compared to Alive are shaded. $\overline{attr}$ represents a list of attributes with each instruction.

3 Alive-FP Domain-Specific Language

Alive-FP is a domain-specific language that adds support for FP operations in LLVM to Alive. Alive-FP is a ground-up rewrite of Alive to simplify the addition of FP reasoning along with other extensible features. Alive-FP uses the FPA theory [4] in SMT-LIB to reason about the correctness of optimizations. An Alive-FP optimization may combine integer and FP operations. Figure 2 lists the new FP types, instructions, and attributes in Alive-FP.

FP types. LLVM has six different floating point types: `half`, `float`, `double`, `fp128`, `x86_fp80` (x86 extended float), and `ppc_fp128` (PowerPC double double). Alive-FP supports five of the six floating point types. Alive-FP does not support `ppc_fp128`, as it uses a pair of double-precision values to represent a number and does not correspond directly to an SMT FP sort. Among the FP types supported by Alive-FP, `half` (k=5, n=10), `float` (k=8, n=23), `double` (k=11, n=52), and `fp128` (k=15, n=112) correspond directly to SMT floating-point sorts, which are determined by the width in bits for the exponent and significand. We treat `x86_fp80` as k=15, n=64, except when its exact bit representation is significant.

FP instructions. LLVM has twelve FP instructions in the LLVM IR. Alive-FP augments Alive with these twelve LLVM instructions dealing with FP values. The FP instructions can be classified into three categories: binary operators, conversion instructions, and a comparison instruction. All LLVM FP instructions are polymorphic over FP types, with the conversion operators imposing some additional constraints. Alive-FP's `select` instruction is polymorphic over all "first-class" types, which include integer and FP types.

Binary operators. Alive-FP supports the five binary arithmetic operators in LLVM for FP computation: `fadd`, `fsub`, `fmul`, `fdiv`, and `frem`, which implement addition, subtraction, multiplication, division, and taking the remainder, respectively. LLVM's `frem` instruction differs from `fpRem` in the IEEE standard and the FPA theory. LLVM's `frem` is similar to the `fmod` function in the C library. Given two floating-point values x and y, the `frem` instruction in LLVM (hence, Alive-FP) calculates x/y and truncates the fractional part. Let's call the resultant value n. Then, `frem x, y` returns $x - ny$.

Conversion operators. LLVM includes two instructions for converting between FP types, `fpext` and `fptrunc`, and four instructions for converting to or from signed and unsigned integers, `fptosi`, `fptoui`, `sitofp`, and `uitofp`. Alive-FP supports all these instructions. The `fpext` instruction promotes a value to a larger FP type, while `fptrunc` demotes a value to a smaller one. The `fptosi` and `fptoui` instructions first discard the fractional part of the input (*i. e.*, rounding to integer towards zero), then map the result to a signed or unsigned integer value, respectively. Similarly, `sitofp` and `uitofp` convert signed and unsigned integers to FP values. Alive-FP also supports the `bitcast` instruction between FP and integer types with the same representation size in bits.

Conversions to floating point can fail in two general ways: they may be inexact, meaning the value being converted falls between two finite values representable in the target type, or they may be out-of-range, meaning the value falls between the largest (or smallest) finite representable value and infinity. Further, LLVM does not specify rounding mode for all instructions.

Comparison instruction. LLVM provides a `fcmp` instruction for comparing FP values, which supports sixteen comparison predicates (see Figure 2). An example is `fcmp uge %x, %y`, where `uge` is one of the sixteen comparison predicates in LLVM, `%x` and `%y` are operands to the `fcmp` instruction. The sixteen predicates are derived from four primitive predicates: *unordered*, which is true if either operand is NaN, and ordered equality, greater-than, and less-than, which are true when neither operand is NaN and the respective condition holds between `%x` and `%y`. The FP comparison predicate `uge` is true if the operands are unordered or equal or the first operand is greater than the second. All sixteen predicates can be translated using combinations of operations in FPA theory.

FP instruction attributes. LLVM defines five FP related instruction attributes that may appear on FP instructions. These are also called fast-math attributes. When present, the optimizer may make certain assumptions about the inputs to these instructions enabling more extensive code transformations.

The no-NaNs attribute, `nnan`, permits the optimizer to assume that the arguments and the result of an instruction will not be NaN. In particular, the LLVM language reference manual states that an operation with a NaN argument or that produces a NaN result is an undefined value (but not undefined behavior). However, it is unclear whether it is an `undef` value or a poison value.

The no-infinities attribute, `ninf`, is similar to `nnan`, except applying to ∞ and $-\infty$. The no-signed-zeros attribute, `nsz`, permits the optimizer to ignore the difference between positive and negative zero. The allow reciprocal attribute, `arcp`, permits the optimizer to assume that multiplying by the reciprocal is equivalent to division. Alive-FP does not handle this attribute as there is only one optimization that uses it. The final attribute, `fast`, implies all the others and permits the optimizer to perform optimizations that are possible with real numbers but can result in inaccurate computation with FP arithmetic (*e. g.*, reassociating arithmetic expressions). Alive-FP will report such optimizations to be incorrect, as they do not preserve bitwise precision. Handling `fast` attribute would require reasoning about accuracy loss [17, 18].

FP constants and literals. Alive-FP supports expressing constant FP values in decimal representation. Alive-FP represents negative zero as `-0.0`, Not-a-Number as `nan`, positive-infinity as `inf`, and negative-infinity as `-inf`. Alive-FP also extends Alive's constant language to be polymorphic over integer and FP types, so that the expression `C+1` may describe an integer or a FP value, depending on the context. Alive-FP adds new constant functions to convert between FP types and to obtain information such as the bit width of an FP type's significand.

4 Verification with Alive-FP

The Alive-FP interpreter checks the correctness of an optimization by encoding FP operations into operations in FPA theory in SMT-LIB. Similar to Alive, integer operations are encoded with operations from the bitvector theory. The Alive-FP interpreter checks the correctness of an optimization for each feasible type. In contrast to Alive, Alive-FP enumerates all feasible types with a non-SMT based type checker to reduce the number of SMT queries.

4.1 Overview of Correctness Checking

Given a concrete type instantiation, the Alive-FP interpreter creates the following SMT expressions for each instruction in both the source and the target: (1) the expression v that represents the result of the instruction, (2) the expression δ that represents constraints for the instruction to have defined behavior, (3) the expression ρ that represents constraints for the instruction to produce a non-poison value. Further, to handle `undef` values, the interpreter also creates a set of quantified variables $\mathcal{U}$ for both the source and target. The validity checks will involve universal quantification for the variables from $\mathcal{U}$ in the target and existential quantification for the variables from $\mathcal{U}$ in the source. The definedness constraints and poison-free constraints propagate with data dependencies. Hence, the order of instructions in an Alive-FP optimization is not important. Instructions which do not introduce undefined behavior, poison, or `undef` values use these encodings by default:

$$\delta_x = \bigwedge_{a \in \mathsf{args}(x)} \delta_a, \qquad \rho_x = \bigwedge_{a \in \mathsf{args}(x)} \rho_a, \qquad \mathcal{U}_x = \bigcup_{a \in \mathsf{args}(x)} \mathcal{U}_a,$$

where δ_x, ρ_x, and $\mathsf{args}(x)$ are the definedness constraints, poison-free constraints, and arguments of instruction x, respectively.

The interpreter also generates SMT expressions corresponding to the precondition. Let $\psi \equiv \phi \wedge \delta^s \wedge \rho^s$ where ϕ represents constraints for the precondition, δ^s represents constraints for the source root to be defined, and ρ^s represents the constraints for the source root to be poison-free. A transformation is correct if and only if all of the following constraints hold for the source and target roots.

1. $\forall_{\mathcal{I},\mathcal{U}^t}\ \exists_{\mathcal{U}^s} : \psi \implies \delta^t$
2. $\forall_{\mathcal{I},\mathcal{U}^t}\ \exists_{\mathcal{U}^s} : \psi \implies \rho^t$
3. $\forall_{\mathcal{I},\mathcal{U}^t}\ \exists_{\mathcal{U}^s} : \psi \implies v^s = v^t$

where $\mathcal{I}$ is the set of input variables in the DAG, $\mathcal{U}^t$ is the set of `undef` variables in the target, $\mathcal{U}^s$ is the set of `undef` variables in the source, δ^t represents the constraints for the target to be well-defined, ρ^t represents the constraints for the target to be poison-free, v^s is the value computed by the source, and v^t is the value computed by the target.

Next, we show how to encode three fast-math attributes (`nnan`, `ninf`, and `nsz`) into FPA theory and their interaction with various kinds of undefined behavior in LLVM. We also highlight the challenges in encoding that arise because the LLVM language reference is under-specified about what values are returned when an instruction is not well-defined. Finally, we provide the encoding for FP instructions in FPA theory and their interaction with fast-math attributes and rounding modes, as LLVM does not define rounding modes for all instructions.

4.2 Encoding Fast-Math Attributes

Fast-math attributes may occur on any of the five arithmetic instructions and the `fcmp` instruction. Optimizations are free to make certain assumptions when these attributes are present in an instruction, specifically:

nnan the arguments and result are not NaN
ninf the arguments and result are not positive or negative infinity
nsz there is no distinction between negative and positive zero

The challenging task in encoding these attributes is in their interaction with various kinds of undefined behavior in LLVM. When an instruction with an attribute violates the assumption, does it produce an `undef` value or a poison value? The key difference between the these two interpretations is that a poison value propagates through dependencies whereas an `undef` value does not. The LLVM language reference for FP operations predates the development of poison values and hence, there is no reference to it in the language reference. The LLVM language reference states that optimizations must "retain defined behavior for NaN (or infinite) inputs, but the value of the result is undefined".

Similar attributes for integer operations in LLVM, no-signed wrap (`nsw`), no-unsigned wrap (`nuw`), and `exact`, produce poison values. Given this under-specification in the LLVM language reference manual, we provide two encodings for `nnan` and `ninf` attributes: one using `undef` values and other with poison values. Both encodings have their own advantages and disadvantages. The encoding with poison values is the most permissive.

Encoding for `nnan` and `ninf` with `undef` values. In this interpretation, an instruction with `nnan` returns an `undef` value when an argument or the result is NaN. For an instruction z performing binary operation ($\oplus$) with arguments x and y, we create a fresh `undef` value and add it to the set of quantified variables if either of the operands or the result is `NaN`. These `undef` variables will be appropriately quantified depending on whether they occur in the source or the target in the validity checks.

$$\mathcal{U}_z = \{u\} \cup \mathcal{U}_x \cup \mathcal{U}_y$$
$$v_z = \begin{cases} u & \text{if } \mathsf{isNaN}(v_x) \vee \mathsf{isNaN}(v_y) \vee \mathsf{isNaN}(v_x \oplus v_y) \\ v_x \oplus v_y & \text{otherwise} \end{cases}$$

where u is a fresh variable and isNaN predicate returns true if its argument is NaN. The encoding for infinities with `undef` values is similar except we use a different predicate (*i. e.*, isInf) with infinities.

Encoding `nnan` and `ninf` with poison values. In this interpretation, an instruction with `nnan` (or the `ninf` attribute) returns a poison value when either its arguments or the result is NaN (or infinity). We encode the fact that these result in poison values in the poison-free constraints for the instruction. The poison-free constraints for an instruction z performing a binary operation ($\oplus$) with arguments x and y are given below:

$$\rho_z = \neg(\mathsf{isNaN}(v_x) \vee \mathsf{isNaN}(v_y) \vee \mathsf{isNaN}(v_x \oplus v_y)) \wedge \rho_x \wedge \rho_y$$
$$v_z = v_x \oplus v_y$$

There are multiple advantages in using the poison interpretation. First, the semantics of poison ensures that the poison value propagates along dependencies. In particular, if a value occurs in multiple instructions, some but not all of which have the `nnan` attribute, then an optimization is free to assume that value is not NaN everywhere. Second, it permits more optimizations and is consistent with the treatment of integer attributes, which use poison values. Third, the encoding does not use quantifiers and FPA theory solvers can check validity quickly. A disadvantage of this encoding is that this interpretation can arguably conflict with the LLVM language reference, which is ambiguous. The encoding for the `ninf` attribute with poison values is similar.

Encoding `nsz` attributes. The `nsz` attribute is used to indicate the assumption that the sign of zero does not matter. The LLVM language reference states that optimizations may "treat the sign of a zero argument or result as insignificant" when an instruction has the `nsz` attribute.

We encode the `nsz` attribute by giving zero-valued results an `undef` sign-bit (*i. e.*, the result is an `undef` value that is constrained to be from the set $\{+0, -0\}$).

For a non-division instruction z with arguments x and y performing the binary operation $\oplus$, we create a fresh `undef` variable and add it to the set of quantified variables. The result is constrained to choose either $+0.0$ or -0.0 based on the `undef` value. These variables to represent `undef` values will be appropriately quantified in the validity checks.

$$\mathcal{U}_z = \{b\} \cup \mathcal{U}_x \cup \mathcal{U}_y$$
$$v_z = \begin{cases} 0 \text{ if } \mathsf{isZero}(v_x \oplus v_y) \land b \\ -0 \text{ if } \mathsf{isZero}(v_x \oplus v_y) \land \neg b \\ v_x \oplus v_y \text{ otherwise} \end{cases}$$

where b is a fresh boolean variable. It is not necessary to examine the arguments of the instruction with the encoding for `nsz` as the sign of any input zero is only significant when the result is also zero.

The encoding is a bit more involved for the division instruction, as the sign of a zero input may affect a non-zero result. Given `z = fdiv x, y`, the result will be ± 0 if x is zero and y non-zero and $\pm\infty$ if y is zero and x non-zero. Other results are unaffected by signed zeros and can be calculated normally.

$$\mathcal{U}_z = \{b\} \cup \mathcal{U}_x \cup \mathcal{U}_y$$
$$v_z = \begin{cases} 0 \text{ if } \mathsf{isZero}(v_x) \land \neg\mathsf{isZero}(v_y) \land b \\ -0 \text{ if } \mathsf{isZero}(v_x) \land \neg\mathsf{isZero}(v_y) \land \neg b \\ \infty \text{ if } \neg\mathsf{isZero}(v_x) \land \mathsf{isZero}(v_y) \land b \\ -\infty \text{ if } \neg\mathsf{isZero}(v_x) \land \mathsf{isZero}(v_y) \land \neg b \\ v_x \div v_y \text{ otherwise} \end{cases}$$

where b is a fresh boolean variable.

Attributes with the `fcmp` instruction. Although `fcmp` accepts the same fast-math attributes as the binary operators, only `nnan` and `ninf` have any effect, because it does not return an FP value. Aside from the type of the `undef` value (for the `undef` encoding), attributes on `fcmp` can be encoded similarly to the binary operators.

4.3 Encoding FP Arithmetic Instructions

LLVM has five FP arithmetic instructions: `fadd`, `fsub`, `fmul`, `fdiv`, and `frem`. The first four translate directly to corresponding SMT FPA operations, aside from the presence of rounding modes. LLVM does not specify what rounding modes to use when performing arithmetic, nor does it provide any standard way for programs to access or determine the rounding mode.

Handling rounding mode with instructions. Alive-FP, by default, performs all arithmetic using a default rounding mode which is chosen by the user at the command line when the tool is run. In most cases, the choice of rounding mode does not affect correctness. Exceptions include optimizations involving zero-valued results, as the RTN (round-to-negative) rounding mode produces -0.0 for many operations that would produce 0.0 under other modes. The choice of rounding mode can also affect whether some out-of-range values round to the largest (or smallest) finite value, or to positive (or negative) infinity.

Alive-FP could also provide an option to make the rounding mode an `undef` value for each instruction, effectively returning a two-valued set (*i. e.*, round up or round down) for any imprecise operation. This interpretation of rounding rules out some existing optimizations as they produce different results depending on the rounding mode chosen. It is unclear whether the complexity of this interpretation is outweighed by any semantic benefit.

Handling the `frem` instruction. The `frem` instruction in LLVM does not correspond to the IEEE floating-point remainder operation, as it requires the remainder to have the same sign as the dividend. LLVM's `frem` corresponds to the `fmod` in the C library. In version 4.4.1, Z3's `fpRem` operation differs from the specification in the SMT-FPA theory and implements the `fmod` function in the C library, which we have reported as a bug [19]. Hence, we encode LLVM's `frem` directly using Z3's `fpRem` operation.

Implementing LLVM's `frem` using a correct SMT-FPA `fpRem` is relatively straightforward, involving some sign manipulation for negative dividends and addition.

$$frem(x, y) = \begin{cases} z \text{ if } \mathsf{isPos}(x) \wedge \mathsf{isPos}(z) \\ |y| + z \text{ if } \mathsf{isPos}(x) \wedge \mathsf{isNeg}(z) \\ -z \text{ if } \mathsf{isNeg}(x) \wedge \mathsf{isPos}(z) \\ -(|y| + z) \text{ if } \mathsf{isNeg}(x) \wedge \mathsf{isNeg}(z) \end{cases}$$
$$z = \mathsf{fpRem}(|x|, |y|)$$

4.4 Encoding Floating-Point Comparison

LLVM's `fcmp` instruction takes a condition code and two floating-point values, and returns a 1-bit integer, indicating whether the comparison is true. The condition code indicates which comparison is to be performed. LLVM includes ordered and unordered versions of the usual equality and inequality tests (see Figure 2). The ordered tests always fail if one or more argument is NaN, and the unordered tests always succeed if one or more argument is NaN. These are encoded using SMT-FPA comparisons in the obvious manner.

4.5 Encoding Conversion Instructions

LLVM (hence, Alive-FP) has the following conversion instructions: `fpext`, `fptrunc`, `fptosi`, `fptoui`, `sitofp`, and `uitofp`. LLVM's language reference does not state whether these operations result in either an `undef` or a poison value in the case of imprecise conversions. Next, we describe our encoding of these operations.

Encoding `fpext` instruction. The `fpext` instruction promotes a floating-point value to a larger type. All floating-point types are strict supersets of smaller types, so `fpext` is always defined. Alive-FP encodes `fpext` as:

$$v_z = \mathsf{fpToFP}(r, v_x, \tau_z)$$

where r is the default rounding mode (the choice is irrelevant), τ_z is the target type, and fpToFP in the FPA theory converts either a FP value or a signed bitvector value to the specified FP type with the specified rounding mode.

Encoding `fptrunc` instruction. The `fptrunc` instruction demotes a floating-point value to a smaller type. Not all values in the larger type are exactly representable in the smaller type. These fall into two categories. A conversion is *imprecise* if the exponent can be represented in the target type, but least significant bits of the source significand are non-zero. LLVM requires the result of an imprecise conversion to be one of the two representable values closest to the source value, but leaves the choice undefined. This can be interpreted as rounding, according to (a) a fixed rounding mode (*e. g.*, RNE — round to the nearest representable number with ties to even), (b) an arbitrary rounding mode, or (c) an undefined choice between the two nearest values. By default, Alive-FP rounds using the current rounding mode in this case.

A conversion is *out of range* if the exponent cannot be represented in the target type. LLVM states that the result is undefined. This may be interpreted as an `undef` or a poison value. We provide encodings for both interpretations.

Encoding out-of-range truncations with `undef` values. In this interpretation, we handle out-of-range conversions by creating fresh `undef` variables that are appropriately quantified.

$$\begin{aligned}\mathcal{U}_z &= \{u\} \cup \mathcal{U}_x\\ v_z &= \begin{cases}\mathsf{fpToFP}(r, v_x, \tau_z) & \text{if } v_x \in \tau_z\\ u & \text{otherwise}\end{cases}\end{aligned}$$

where r is the default rounding mode, τ_z is the target type, and u is a fresh variable. We write $v_x \in \tau_z$ to indicate that v_x is exactly representable in τ_z.

Encoding out-of-range truncations with poison values. In this interpretation, we encode out-of-range conversions in the poison-free constraints.

$$\begin{aligned}\rho_z &= v_x \in \tau_z \land \rho_x\\ v_z &= \mathsf{fpToFP}(r, v_x, \tau_z)\end{aligned}$$

where τ_z is the target type, r is the current rounding mode, and ρ_x are the poison-free constraints for x.

Encoding `fptosi` instruction. The `fptosi` instruction converts a floating-point value to a bitvector, interpreted as a signed value. Any fractional part of the value is truncated (*i. e.*, rounded towards zero). If the resulting value is outside the range of the target type, the result is undefined. We provide encodings both as an `undef` value and a poison value to represent values outside the range of the target type.

Encoding out-of-range `fptosi` conversions with a `undef` value. We create a fresh **undef** variable, which will be appropriately quantified, when the value to be converted is beyond the range.

$$\begin{aligned}\mathcal{U}_z &= \{u\} \cup \mathcal{U}_x\\ v_z &= \begin{cases}\mathsf{fpToSBV}(\mathsf{RTZ}, v_x, \tau_z) & \text{if } smin(\tau_z) \le v_x \le smax(\tau_z)\\ u & \text{otherwise}\end{cases}\end{aligned}$$

where u is a fresh variable, τ_z is the target type, and fpToSBV in the FPA theory converts a floating point number to signed bitvector similar to the semantics of `fptosi` instruction. Constant functions $smin(\tau_z)$ and $smax(\tau_z)$ return the largest and smallest signed values for a type τ_z. We use the rounding mode RTZ (round towards zero) because the LLVM language reference manual explicitly specifies it.

Encoding out-of-range `fptosi` conversions with a poison value. The encoding is similar to other instructions except the poison-free constraints use the constant functions *smin* and *smax*.

$$\begin{aligned}\rho_z &= smin(\tau_z) \leq v_x \leq smax(\tau_z) \wedge \rho_x \\ v_z &= \mathsf{fpToSBV}(\mathsf{RTZ}, v_x, \tau_z)\end{aligned}$$

where τ_z is the target type.

Encoding the `fptoui` instruction. The `fptoui` instruction converts a floating-point value to a bitvector, interpreted as an unsigned value. The encoding is similar to the `fptosi` instruction except we use the constant functions *umin* and *umax* (that return unsigned minimum and unsigned maximum value for a type), and fpToUBV in the FPA theory to convert the value to an unsigned bitvector.

Encoding the `sitofp` instruction. The `sitofp` instruction converts a bitvector, interpreted as a signed integer, to a floating-point value. As with `fptrunc`, such a conversion may be exact, imprecise, or out-of-range. Alive-FP handles imprecise conversions by rounding according to the current rounding mode. Alive-FP provides both `undef` and poison interpretations for out-of-range conversions.

Encoding out-of-range `sitofp` conversions with `undef`. The encoding is similar to `fptrunc` except we use the constant functions $fmin$ and $fmax$ that provide the largest and smallest finite values for a given floating point type.

$$\begin{aligned}\mathcal{U}_z &= \{u\} \cup \mathcal{U}_x \\ v_z &= \begin{cases}\mathsf{sbvToFP}(r, v_x, \tau_z) & \text{if } fmin(\tau_z) \leq v_x \leq fmax(\tau_z) \\ u & \text{otherwise}\end{cases}\end{aligned}$$

where u is a fresh variable, τ_z is the target type, and r is the current rounding mode.

Encoding out-of-range `sitofp` conversions with poison. Similar to the encoding for `fptrunc`, we encode out-of-range conversions with `sitofp` as a poison value that result in an additional poison-free constraint.

$$\begin{aligned}\rho_z &= fmin(\tau_z) \leq v_x \leq fmax(\tau_z) \wedge \rho_x \\ v_z &= \mathsf{sbvToFP}(r, v_x, \tau_z)\end{aligned}$$

where r is the current rounding mode and τ_z is the target type.

Encoding the `uitofp` instruction. The `uitofp` instruction converts a bitvector, interpreted as an unsigned integer, to a floating-point value. This

```
%r = fmul nnan nsz %x, -1
=>
%r = fsub nnan nsz -0.0, %x
```

(a) An example optimization

$$P = \mathsf{isNaN}(x) \lor \mathsf{isNaN}(-1) \lor \mathsf{isNaN}(x \times -1)$$
$$Q = \mathsf{isNaN}(-0) \lor \mathsf{isNaN}(x) \lor \mathsf{isNaN}(-0 - x)$$

(b) The non-NaN conditions, used below

$$\begin{aligned}
\mathcal{U}^s &= \{u_1, b_1\}\\
\delta^s &= \top\\
\rho^s &= \top\\
v^s &= \begin{cases} u_1 \text{ if } P\\ 0 \text{ if } \neg P \land \mathsf{isZero}(x \times -1) \land b_1\\ -0 \text{ if } \neg P \land \mathsf{isZero}(x \times -1) \land \neg b_1\\ x \times -1 \text{ if } \neg P \land \neg\mathsf{isZero}(x \times -1)\end{cases}\\
\mathcal{U}^t &= \{u_2, b_2\}\\
\delta^t &= \top\\
\rho^t &= \top\\
v^t &= \begin{cases} u_2 \text{ if } Q\\ 0 \text{ if } \neg Q \land \mathsf{isZero}(-0 - x) \land b_2\\ -0 \text{ if } \neg Q \land \mathsf{isZero}(-0 - x) \land \neg b_2\\ -0 - x \text{ if } \neg Q \land \neg\mathsf{isZero}(-0 - x)\end{cases}
\end{aligned}$$

value check: $\forall x\, u_2\, b_2, \exists u_1\, b_1, v^s = v^t$

(c) The `undef` encoding

$$\begin{aligned}
\mathcal{U}^s &= \{b_1\}\\
\delta^s &= \top\\
\rho^s &= \neg P\\
v^s &= \begin{cases} 0 \text{ if } \mathsf{isZero}(x \times -1) \land b_1\\ -0 \text{ if } \mathsf{isZero}(x \times -1) \land \neg b_1\\ x \times -1 \text{ if } \neg\mathsf{isZero}(x \times -1)\end{cases}\\
\mathcal{U}^t &= \{b_2\}\\
\delta^t &= \top\\
\rho^t &= \neg Q\\
v^t &= \begin{cases} 0 \text{ if } \mathsf{isZero}(-0 - x) \land b_2\\ -0 \text{ if } \mathsf{isZero}(-0 - x) \land \neg b_2\\ -0 - x \text{ if } \neg\mathsf{isZero}(-0 - x)\end{cases}
\end{aligned}$$

poison check: $\forall x\, b_2, \exists b_1, \rho^s \implies \rho^t$

value check: $\forall x\, b_2, \exists b_1, \rho^s \implies v^s = v^t$

(d) The poison encoding

Fig. 3. Illustration of the encodings and the validity checks generated by Alive-FP for the optimization shown in (a). To simplify the exposition, we use the constraints in (b) in the examples. The constraints and the validity checks generated with the `undef` interpretation for `nnan` attribute is shown in (c). The constraints and the validity checks generated with the poison interpretation for the `nnan` attribute is shown in (d). The `fmul` instruction is always defined. Hence, the definedness condition (δ^s for source and δ^t for the target is true ($\top$). In the undef interpretation in (c), there are no poison values, hence poison-free conditions are set to true (*i. e.*, $\rho^s = \top$ and $\rho^t = \top$).

is handled analogously to `sitofp`, but using the corresponding unsigned conversion, $\mathsf{ubvToFP}$ in FPA theory.

Encoding `bitcast` instruction with floating point operations. The bitcast instruction converts a value from one type to another without changing any bits. Thus, its source and target types must have the same bit width. When converting an integer to a floating-point value, Alive-FP uses SMT-FPA's fpToFP, which converts a bit vector to a floating-point value directly when called with two arguments.

$$v_z = \mathsf{fpToFP}(v_x, \tau_z)$$

When converting a floating-point value to an integer, we use Z3's $\mathsf{fpToIEEEBV}$, a non-standard addition to the FPA theory.

$$v_z = \mathsf{fpToIEEEBV}(v_x, \tau_z)$$

This has the limitation of only producing one bit pattern for NaNs. An alternative would be returning an `undef` value restricted so that its conversion to floating point is equal to v_x.

Illustration of correctness checking. Figure 3(a) presents a simple Alive-FP optimization. Figure 3(c) and Figure 3(d) present the encoding and the validity checks generated with the `undef` and poison encoding for the `nnan` attribute.

5 Evaluation

This section describes our prototype, the optimizations that we translated from LLVM to Alive-FP, new bugs discovered, and the time taken for verification.

Prototype. Alive-FP is a ground-up rewrite of the Alive infrastructure, designed with the goal of extending the language and the semantics. The Alive-FP interpreter is written in Python and uses Z3-4.4.1 for solving SMT queries. Alive-FP supports a subset of the features in Alive—enough to verify the Alive suite of optimizations—along with the new FP support. Specifically, Alive-FP does not support memory operations and cycle detection with composition [20]. Unlike Alive, which uses SMT queries to find feasible type instantiations, Alive-FP enumerates all feasible type models using an unification-based type checker.

When we were designing FP support in Alive-FP, we discovered that Alive's treatment of `undef` was incorrect in some scenarios where a single `undef` was referenced more than once [21]. Alive-FP addresses this problem by generating fresh quantified variables for each reference to an `undef` value or an instruction that may produce `undef` values. Alive-FP is open source [13].

Translation of LLVM optimizations to Alive-FP. We translated all the bit-precise FP optimizations, which do not result in loss of accuracy, in LLVM-3.8 to Alive-FP. In total, we generated a total of 104 Alive-FP optimizations. Among these 104 Alive-FP optimizations, 48 optimizations were from `InstSimplify`, 12 are FP mul/div/rem optimizations, 37 are FP compare optimizations, 1 is a cast optimization, and 6 are FP add/sub optimizations. Certain optimizations in LLVM are expressed as multiple Alive-FP optimizations, in particular optimizations involving `fcmp` conditions or conditional flags in the target.

New LLVM bugs reported with Alive-FP. We discovered and reported seven wrong FP optimizations in LLVM [8–12], which are listed in Figure 4. Among these reported bugs, PR26746 [8] incorrectly changes the sign of zero and has already been fixed. The sign of the zero is important for complex elementary functions [22], where having a single zero can introduce discontinuities. After we reported bug PR26862-1 in Figure 4, a developer pointed us to a discussion in LLVM developers mailing list [23] that alluded to the argument that an `undef` value in the source can be a signaling NaN [6] and therefore the source likely exhibits undefined behavior. This argument is flawed because not all hardware supports traps for signaling NaNs, and because it effectively erases the distinction between `undef` and undefined behavior for FP values, which is explicit in the LLVM language reference. The optimization PR27151 [11] in Figure 4 is wrong

```
Name: PR26746
Pre: C == 0.0
%1 = fsub -0.0, %x
%r = fsub C, %1
=>
%r = %x
```

```
Name: PR26862-1
%r = fdiv undef, %x
=>
%r = undef

Name: PR26862-2
%r = fdiv %x, undef
=>
%r = undef
```

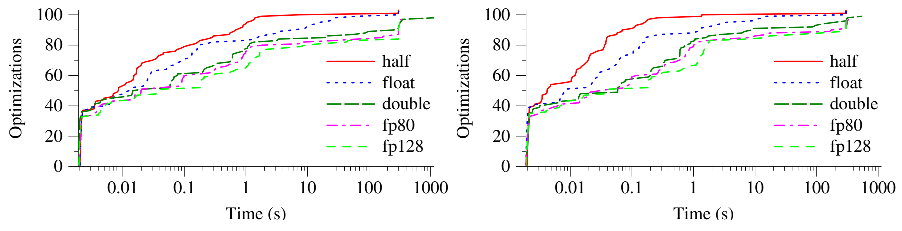
```
Name: PR26863-1
%r = frem undef, %x
=>
%r = undef

Name: PR26863-2
%r = frem %x, undef
=>
%r = undef
```

```
Name: PR27151
Pre: C == 0.0
%y = fsub nnan ninf C, %x
%z = fadd %y, %x
=>
%z = 0
```

```
Name: PR27153
Pre: sitofp(C0) == C && \
  WillNotOverflowSignedAdd(%a, C0)
%x = sitofp %a
%r = fadd %x, C
=>
C0 = fptosi(C)
%y = add nsw %a, C0
%r = sitofp %y
```

Fig. 4. Seven wrong optimizations discovered by Alive-FP. In PR26746 [8], the optimization is wrong because when $x = -0.0$ and $C = 0$, the source always evaluates to 0 but target evaluates to -0. In PR26862-1 [9], the optimization is wrong because if $x = \pm 0$, the only possible return values for the source are $\pm\infty$ and NaN, but the target can produce any floating-point value. In PR26862-2 [9], the optimization is wrong because if $x = \pm 0$, the only possible return values for the source are ± 0 and NaN, but the target may return any value. In PR26863-1 [10], the optimization is wrong because if $x = \pm 0$, the only possible return value for the source is NaN, but the target may return any value. In fact, for any finite, positive x, we will have $r < x$ in the target. In PR26863-2 [10], the optimization is wrong because for a finite x we have $|r| < |x|$ for the source, but the target may return any value. The optimization in PR27151 [11] is wrong in the `undef` interpretation but correct in the `poison` interpretation for `nnan`. With `undef` interpretation, when x is NaN, then y will be an `undef` value. The value z in the source will always be NaN, because any value added to NaN is NaN but the target is 0. In the poison interpretation, we can assume x is not NaN, because that would result in z being a poison value. Because z can never be visibly used without causing undefined behavior, we are free to replace it with 0 in that case. The buggy optimization PR27153 [12] reassociates an integer-to-floating point conversion and a constant add, when LLVM can show that the constant C is exactly representable in the type of a and the sum $a + C$ is also representable in that type (*i. e.*, computing it will not overflow). The optimization fails if a cannot be exactly represented in the target type but $a + C$ can be (this could occur if a and C have opposite signs). In such a case, the source will lose precision due to rounding, but the target will not, resulting in changed behavior.

depending on the interpretation used for `nnan` and `ninf` attributes. The optimization is wrong under the `undef` interpretation but is correct under the poison interpretation. The optimization PR27153 [12] in Figure 4 is wrong because the



(a) `undef` encoding (b) Poison encoding

Fig. 5. The number of optimizations solvable in a given time with the `undef` encoding and the poison encoding. The x-axis is in log-scale. We could not verify some of the optimizations with larger FP types because the solver returned `unknown` result, ran out of memory, or did not complete.

target has a higher precision than the source, which violates the expectation of bitwise similarity.

Time taken to verify optimizations. Figure 5 reports the number of optimizations that can be verified in a given amount of time for different FP types with the `undef` and the poison encoding. We conducted this experiment by specializing the FP type on a x86-64 2.6GHz Haswell machine with 16GB main memory. We were not able to verify 3 optimizations with any FP type because the solver did not complete. Figure 5 illustrates that the poison encoding is faster than the `undef` encoding. Further, the smaller FP types are verified quickly.

6 Related Work

There is significant prior research on checking, estimating, and/or improving the accuracy of FP operations [17, 18, 24–33], which is orthogonal to Alive-FP. Our work is also related to analyses and formal verification of bit-precise FP computations [30, 34–36]. In the context of verified compilers, CompCert [37] provides semantics to FP operations and performs mechanized verification of bit-precise FP computations [38]. In contrast, Alive-FP performs automated verification by encoding into FPA theory and takes into account the undefined behavior in the LLVM compiler.

Alive-FP is also related to prior DSLs for specifying optimizations [39–41], which typically do not address FP operations. Alive-FP builds on our prior work Alive [3], which focused on integer optimizations. Concurrent to Alive-FP, LifeJacket [42] is initial work that also adds FP reasoning to Alive. In contrast to Alive-FP, LifeJacket does not explore the various semantic interpretations for attributes or instructions. LifeJacket primarily uses `undef` values for any undefined behavior. Its encoding for the `nsz` attribute is likely restrictive and can incorrectly rule out optimizations. Also, it does not support the `bitcast` instruction or the `x86_fp80` and `fp128` types.

7 Conclusion

We have presented Alive-FP, a domain-specific language to express and verify bit-precise FP optimizations in LLVM. We have proposed multiple encodings for FP operations and fast-math attributes (`nsz`, `nnan`, and `ninf`) to account for the ambiguity in the LLVM language reference. Alive-FP provides a comprehensive semantic treatment of FP operations and its interaction with undefined behavior in LLVM. Alive-FP is a first step in the automated verification of bit-precise FP computations in the LLVM compiler.

Acknowledgments

We thank Vinod Ganapathy, Thomas Wahl, and the reviewers for their feedback on this paper. This paper is based on work supported in part by NSF CAREER Award CCF–1453086, a sub-contract of NSF Award CNS–1116682, a NSF Award CNS–1441724, a Google Faculty Award, and gifts from Intel Corporation.

References

1. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 283–294. PLDI, ACM (2011)
2. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 216–226. PLDI (2014)
3. Lopes, N., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with Alive. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 22–32. PLDI, ACM (2015)
4. Brain, M., Tinelli, C., Rümmer, P., Wahl, T.: An automatable formal semantics for IEEE-754 floating-point arithmetic. In: Proceedings of the 22nd IEEE Symposium on Computer Arithmetic. pp. 160–167. ARITH, IEEE (Jun 2015)
5. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS (2008)
6. IEEE standard for floating-point arithmetic. IEEE 754-2008, IEEE Computer Society (Aug 2008)
7. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys 23(1), 5–48 (Mar 1991)
8. Menendez, D.: LLVM bug 26746 InstructionSimplify turns 0.0 to -0.0. `https://llvm.org/bugs/show_bug.cgi?id=26746`, retrieved 2016-04-16
9. Menendez, D.: LLVM bug 26862 InstructionSimplify broadens undef when simplifying frem. `https://llvm.org/bugs/show_bug.cgi?id=26862`, retrieved 2016-04-16
10. Menendez, D.: LLVM bug 26863 InstructionSimplify broadens undef when simplifying fdiv. `https://llvm.org/bugs/show_bug.cgi?id=26863`, retrieved 2016-04-16

11. Menendez, D.: LLVM bug 27151 InstructionSimplify turns NaN to 0.0. https://llvm.org/bugs/show_bug.cgi?id=27151, retrieved 2016-04-16
12. Menendez, D.: LLVM bug 27153 InstCombine changes results by reassociating addition and sitofp. https://llvm.org/bugs/show_bug.cgi?id=27153, retrieved 2016-04-16
13. Menendez, D., Nagarakatte, S.: Alive-NJ. https://github.com/rutgers-apl/alive-nj, retrieved 2016-04-16
14. Lattner, C.: What every C programmer should know about undefined behavior. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html (May 2011), retrieved 2016-04-16
15. Regehr, J.: A guide to undefined behavior in C and C++. http://blog.regehr.org/archives/213 (Jul 2010), retrieved 2016-04-16
16. Wang, X., Zeldoivch, N., Kaashoek, M.F., Solar-Lezama, A.: Towards optimization-safe systems: Analyzing the impact of undefined behavior. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles. pp. 260–275. SOSP, ACM (Nov 2013)
17. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 1–11. PLDI, ACM (Jun 2015)
18. Solovyev, A., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. In: Proceedings of the 20th International Symposium on Formal Methods. pp. 532–550. FM, Springer (Jun 2015)
19. Menendez, D.: FPA remainder does not match SMT-FPA semantics. https://github.com/Z3Prover/z3/issues/561, retrieved 2016-04-16
20. Menendez, D., Nagarakatte, S.: Termination-checking for LLVM peephole optimizations. In: Proceedings of the 38th International Conference of Software Engineering. pp. 191–202. ICSE (May 2016)
21. Menendez, D.: Incorrect undef semantics. https://github.com/nunoplopes/alive/issues/31, retrieved 2016-04-17
22. Kahan, W.: Branch cuts for complex elementary functions, or much ado about nothing's sign bit. In: Proceedings of the Joint IMA/SIAM Conference on the State of the Art in Numerical Analysis Held at the UN. pp. 165–211 (1987)
23. Anderson, O.: Re: [llvmdev] bug 16257 - fmul of undef ConstantExpr not folded to undef. http://lists.llvm.org/pipermail/llvm-dev/2014-August/076225.html (Aug 2014), retreived 2016-04-17
24. Ivančić, F., Ganai, M.K., Sankaranarayanan, S., Gupta, A.: Numerical stability analysis of floating-point computations using software model checking. In: Proceedings of the 8th IEEE/ACM International Conference on Formal Methods and Models for Codesign. pp. 49–58. MEMOCODE, IEEE (Jul 2010)
25. Darulova, E., Kuncak, V.: Sound compilation of reals. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 235–248. POPL, ACM, New York, NY, USA (2014)
26. Kinsman, A.B., Nicolici, N.: Finite precision bit-width allocation using SAT-modulo theory. In: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 1106–1111. DATE '09, European Design and Automation Association, 3001 Leuven, Belgium, Belgium (2009)
27. Rubio-González, C., Nguyen, C., Nguyen, H.D., Demmel, J., Kahan, W., Sen, K., Bailey, D.H., Iancu, C., Hough, D.: Precimonious: Tuning assistant for floating-

point precision. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 27:1–27:12. SC '13, ACM, New York, NY, USA (2013)

28. Fu, Z., Bai, Z., Su, Z.: Automated backward error analysis for numerical code. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 639–654. OOPSLA 2015, ACM, New York, NY, USA (2015)
29. Barr, E.T., Vo, T., Le, V., Su, Z.: Automatic detection of floating-point exceptions. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 549–560. POPL '13, ACM, New York, NY, USA (2013)
30. Goubault, E.: Static analyses of the precision of floating-point operations. In: Proceedings of the 8th International Symposium on Static Analysis. pp. 234–259. SAS, Springer (2001)
31. de Dinechin, F., Lauter, C.Q., Melquiond, G.: Assisted verification of elementary functions using Gappa. In: Proceedings of the 2006 ACM Symposium on Applied Computing. pp. 1318–1322. SAC, ACM (2006)
32. Brain, M., DSilva, V., Griggio, A., Haller, L., Kroening, D.: Interpolation-based verification of floating-point programs with abstract CDCL. In: Proceedings of the 20th International Symposium on Static Analysis. pp. 412–432. SAS, Springer (Jun 2013)
33. Goubault, E., Putot, S., Baufreton, P., Gassino, J.: Static analysis of the accuracy in control systems: Principles and experiments. In: Revised Selected Papers from the 12th International Workshop on Formal Methods for Industrial Critical Systems, pp. 3–20. FMICS, Springer (2007)
34. Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems (TOPLAS) 30(3), 12:1–12:41 (May 2008)
35. Martel, M.: Semantics-based transformation of arithmetic expressions. In: Proceedings of the 14th International Symposium on Static Analysis. pp. 298–314. SAS, Springer (2007)
36. Harrison, J.: Floating point verification in HOL. In: Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications. pp. 186–199. Springer (Sep 1995)
37. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (Jul 2009)
38. Boldo, S., Jourdan, J.H., Leroy, X., Melquiond, G.: A formally-verified C compiler supporting floating-point arithmetic. In: Proceedings of the 21st IEEE Symposium on Computer Arithmetic. pp. 107–115. ARITH, IEEE (Apr 2013)
39. Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 364–377. POPL (2005)
40. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 327–337. PLDI (2009)
41. Buchwald, S.: Optgen: A generator for local optimizations. In: Proceedings of the 24th International Conference on Compiler Construction. pp. 171–189. CC (2015)
42. Nötzli, A., Brown, F.: LifeJacket: Verifying precise floating-point optimizations in LLVM. `http://arxiv.org/pdf/1603.09290v1.pdf`, retrieved 2016-04-04