Icing: supporting fast-math style optimizations in a verified compiler

Downloaded from: https://research.chalmers.se, 2026-10-14 16:33 UTC

Citation for the original published paper (version of record):
Becker, H., Darulova, E., Myreen, M. et al (2019). Icing: supporting fast-math style optimizations in a verified compiler. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 11562 LNCS: 155-173. http://dx.doi.org/10.1007/978-3-030-25543-5_10

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Icing: Supporting Fast-Math Style Optimizations in a Verified Compiler

Heiko Becker[1(✉)], Eva Darulova[1], Magnus O. Myreen[2], and Zachary Tatlock[3]

[1] MPI-SWS, Saarland Informatics Campus (SIC), Saarbrücken, Germany
{hbecker,eva}@mpi-sws.org
[2] Chalmers University of Technology, Gothenburg, Sweden
myreen@chalmers.se
[3] University of Washington, Seattle, USA
ztatlock@cs.washington.edu

Abstract. Verified compilers like CompCert and CakeML offer increasingly sophisticated optimizations. However, their deterministic source semantics and strict IEEE 754 compliance prevent the verification of "fast-math" style floating-point optimizations. Developers often selectively use these optimizations in mainstream compilers like GCC and LLVM to improve the performance of computations over noisy inputs or for heuristics by allowing the compiler to perform intuitive but IEEE 754-unsound rewrites.

We designed, formalized, implemented, and verified a compiler for Icing, a new language which supports selectively applying fast-math style optimizations in a verified compiler. Icing's semantics provides the first formalization of fast-math in a verified compiler. We show how the Icing compiler can be connected to the existing verified CakeML compiler and verify the end-to-end translation by a sequence of refinement proofs from Icing to the translated CakeML. We evaluated Icing by incorporating several of GCC's fast-math rewrites. While Icing targets CakeML's source language, the techniques we developed are general and could also be incorporated in lower-level intermediate representations.

Keywords: Compiler verification · Floating-point arithmetic · Optimization

1 Introduction

Verified compilers formally guarantee that compiled machine code behaves according to the specification given by the source program's semantics. This stringent requirement makes verifying "end-to-end" compilers for mainstream languages challenging, especially when proving sophisticated optimizations that developers rely on. Recent verified compilers like CakeML [38] for ML and

Z. Tatlock—This work was supported in part by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

© The Author(s) 2019
I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11562, pp. 155–173, 2019.
https://doi.org/10.1007/978-3-030-25543-5_10

CompCert [24] for C have been steadily verifying more of these important optimizations [39–41]. While the gap between verified compilers and mainstream alternatives like GCC and LLVM has been shrinking, so-called "fast-math" floating-point optimizations remain absent in verified compilers.

Fast-math optimizations allow a compiler to perform rewrites that are often intuitive when interpreted as real-valued identities, but which may not preserve strict IEEE 754 floating-point behavior. Developers selectively enable fast-math optimizations when implementing heuristics, computations over noisy inputs, or error-robust applications like neural networks—typically at the granularity of individual source files. The IEEE 754-unsound rewrites used in fast-math optimizations allow compilers to perform strength reductions, reorder code to enable other optimizations, and remove some error checking [1,2]. Together these optimization can provide significant savings and are widely-used in performance-critical applications [12].

Unfortunately, strict IEEE 754 source semantics prevents proving fast-math optimizations correct in verified compilers like CakeML and CompCert. Simple strength-reducing rewrites like fusing the expression $x * y + z$ into a faster and locally-more-accurate fused multiply-add (`fma`) instruction cannot be included in such verified compilers today. This is because `fma` avoids an intermediate rounding and thus may not produce exactly the same bit-for-bit result as the unoptimized code. More sophisticated optimizations like vectorization and loop invariant code motion depend on reordering operations to make expressions available, but these cannot be verified since floating-point arithmetic is not associative. Even simple reductions like rewriting $x - x$ to 0 cannot be verified since the result can actually be `NaN` ("not a number") if x is `NaN`. Each of these cases represent rewrites that developers would often, in principle, be willing to apply manually to improve performance but which can be more conveniently handled by the compiler. Verified compilers' strict IEEE 754 source semantics similarly hinders composing their guarantees with recent tools designed to *improve accuracy* of a source program [14,16,32], as these tools change program behavior to reduce rounding error. In short, developers today are forced to choose between verified compilers and useful tools based on floating-point rewrites.

The crux of the mismatch between verified compilers and fast-math lies in the source semantics: verified compilers implement strict IEEE 754 semantics while developers are intuitively programming against a looser specification of floating-point closer to the reals. Developers currently indicate this perspective by passing compiler flags like `--ffast-math` for the parts of their code written against this looser semantics, enabling mainstream compilers to aggressively optimize those components. Ideally, verified compilers will eventually support such loosened semantics by providing an "approximate real" data type and let the developer specify error bounds under which the compiler could freely apply any optimization that stays within bounds. A good interface to tools for analyzing finite-precision computations [11,16] could even allow independently-established formal accuracy guarantees to be composed with compiler correctness.

As an initial step toward this goal, we present a pragmatic and flexible approach to supporting fast-math optimizations in verified compilers. Our approach follows the implicit design of existing mainstream compilers by providing two complementary features. First, our approach provides fine-grained control over which parts of a program the compiler may optimize under extended floating-point semantics. Second, our approach provides flexible extensions to the floating-point semantics specified by a set of high-level rewrites which can be specialized to different parts of a program. The result is a new nondeterministic source semantics which grants the compiler freedom to optimize floating-point code within clearly defined bounds.

Under such extended semantics, we verify a set of common fast-math optimizations with the simulation-based proof techniques already used in verified compilers like CakeML and CompCert, and integrate our approach with the existing compilation pipeline of the CakeML compiler. To enable these proofs, we provide various *local* lemmas that a developer can prove about their rewrites to ensure *global* correctness of the verified fast-math optimizer. Several challenges arise in the design of this decomposition including how to handle "duplicating rewrites" like distributivity that introduce multiple copies of a subexpression and how to connect context-dependent rewrites to other analyses (e.g., from accuracy-verification tools) via rewrite preconditions. Our approach thus provides a rigorous formalization of the intuitive fast-math semantics developers already use, provides an interface for dispatching proof obligations to formal numerical analysis tools via rewrite preconditions, and enables bringing fast-math optimizations to verified compilers.

In summary, the contributions of this paper are:

- We introduce an extensible, nondeterministic semantics for floating-point computations which allows for fast-math style compiler optimizations with flexible, yet fine-grained control in a language we call *Icing*.
- We implement three optimizers based on Icing: a baseline strict optimizer which provably preserves IEEE 754 semantics, a greedy optimizer, which applies any available optimization, and a conditional optimizer which applies an optimization whenever an (optimization-specific) precondition is satisfied. The code is available at https://gitlab.mpi-sws.org/AVA/Icing.
- We formalize Icing and verify our three different optimizers in HOL4.
- We connect Icing to CakeML via a translation from Icing to CakeML source and verify its correctness via a sequence of refinement proofs.

2 The Icing Language

In this section we define the Icing language and its semantics to support fast-math style optimizations in a verified compiler. Icing is a prototype language whose semantics is designed to be extensible and widely applicable instead of focusing on a particular implementation of fast-math optimizations. This allows us to provide a stable interface as the implementation of the compiler changes, as well as supporting different optimization choices in the semantics, depending on the compilation target.

2.1 Syntax

Icing's syntax is shown in Fig. 1. In addition to arithmetic, let-bindings and conditionals, Icing supports `fma` operators, lists ($[e_1 \ldots]$), projections ($e_1[n]$), and `Map` and `Fold` as primitives. Conditional guards consist of boolean constants (b), binary comparisons ($e_1 \,\square\, e_2$), and an `isNaN` predicate. `isNaN` e_1 checks whether e_1 is a so-called *Not-a-Number* (`NaN`) special value. Under the IEEE 754 standard, undefined operations (e.g., square root of a negative number) produce `NaN` results, and most operations propagate `NaN` results when passed a `NaN` argument. It is thus common to add checks for `NaN`s at the source or compiler level.

$$w\text{: 64-bit floating-point word} \qquad x\text{: String} \qquad n \in \mathbb{N} \qquad b \in \{\text{True}, \text{False}\}$$
$$\diamond \in \{-, \texttt{sqrt}\} \qquad \circ \in \{+, -, *, /\} \qquad \square \in \{<, \leq, =\}$$

$$\begin{aligned} e_1, e_2, e_3 ::= {} & w \mid x \mid [e_1, \ldots] \mid e_1[n] \mid \diamond\, e_1 \mid e_1 \circ e_2 \mid \texttt{fma}(e_1, e_2, e_3) \mid \texttt{opt} : (e_1) \mid \\ & \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \mid \texttt{if}\, c\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \mid \texttt{Map}\, (\lambda x.e_1)\, e_2 \mid \texttt{Fold}\, (\lambda x\, y.e_1)\, e_2\, e_3 \\ c ::= {} & b \mid \texttt{isNaN}\, e_1 \mid e_1 \,\square\, e_2 \mid \texttt{opt} : (c) \end{aligned}$$

Fig. 1. Syntax of Icing expressions

We use the `Map` and `Fold` primitives to show that Icing can be used to express programs beyond arithmetic, while keeping the language simple. Language features like function definitions or general loops do not affect floating-point computations with respect to fast-math optimizations and are thus orthogonal.

The `opt:` scoping annotation implements one of the key features of Icing: floating-point semantics are relaxed only for expressions under an `opt:` scope. In this way, `opt:` provides fine-grained control both for expressions and conditional guards.

2.2 Optimizations as Rewrites

Fast-math optimizations are typically local and syntactic, i.e., peephole rewrites. In Icing, these optimizations are written as $s \to t$ to denote finding any subexpression matching pattern s and rewriting it to t, using the substitution from matching s to instantiate pattern variables in t as usual. The find and replace patterns of a rewrite are terms from the following pattern language which mirrors Icing syntax:

$$p_1, p_2, p_3 ::= w \mid b \mid x \mid \diamond p_1 \mid p_1 \circ p_2 \mid p_1 \,\square\, p_2 \mid \texttt{fma}\,(p_1, p_2, p_3) \mid \texttt{isNaN}\, p_1$$

Table 1 shows the set of rewrites currently supported in our development. While this set does not include all of GCC's fast-math optimizations, it does cover the three primary categories:

- performance and precision improving strength reduction which fuses $x * y + z$ into an `fma` instruction (Rewrite 1)

- reordering based on real-valued identities, here commutativity, and associativity of $+, *$, double negation and distributivity of $*$ (Rewrites 2–5)
- simplifying computation based on (assumed) real-valued behavior for computations by removing `NaN` error checks (Rewrite 6)

A key feature of Icing's design is that each rewrite can be guarded by a *rewrite precondition*. We distinguish *compiler rewrite preconditions* as those that must be true for the rewrite to be correct with respect to Icing semantics. Removing a `NaN` check, for example, can change the runtime behavior of a floating-point program: a previously crashing program may terminate or vice-versa. Thus a `NaN`-check can only removed if the value can never be a `NaN`.

In contrast, an *application rewrite precondition* guards a rewrite that can always be proven correct against the Icing semantics, but where a user may still want finer-grained control. By restricting the context where Icing may fire these rewrites, a user can establish end-to-end properties of their application, e.g., worst-case roundoff error. The crucial difference is that the compiler preconditions must be discharged before the rewrite can be proven correct against the Icing semantics, whereas the application precondition is an additional restriction limiting where the rewrite is applied for a specific application.

A key benefit of this design is that *rewrite preconditions can serve as an interface to external tools* to determine where optimizations may be conditionally applied. This feature enables Icing to address limitations that have prevented previous work from proving fast-math optimizations in verified compilers [5] since "The only way to exploit these [floating-point] simplifications while preserving semantics would be to apply them conditionally, based on the results of a static analysis (such as FP interval analysis) that can exclude the problematic cases." [5] In our setting, a static analysis tool can be used to establish an application rewrite precondition, while compiler rewrite preconditions can be discharged during (or potentially after) compilation via static analysis or manual proof.

This design choice essentially decouples the floating-point static analyzer from the general-purpose compiler. One motivation is that the compiler may perform hardware-specific rewrites, which source-code-based static analyzers would generally not be aware of. Furthermore, integrating end-to-end verification of these rewrites into a compiler would require it to always run a global static analysis. For this reason, we propose an interface which communicates only the necessary information.

Rewrites which duplicate matched subexpressions, e.g., distributing multiplication over addition, required careful design in Icing. Such rewrites can lead to unexpected results if different copies of the duplicated expression are optimized differently; this also complicates the Icing correctness proof. We show how preconditions additionally enabled us to address this challenge in Sect. 4.

Icing optimizes code by folding a list of rewrites over a program `e`:

```
rewrite ([],e) = e
rewrite ((s → t)::rws, e) =
  let e' = if (matches e s) then (app (s → t) e) else e in
  rewrite (rws, e')
```

For rewrite `s→t` at the head of `rws`, `rewrite (rws, e)` checks if `s` matches `e`, applies the rewrite if so, and recurses. Function `rewrite` is used in our optimizers in a bottom-up traversal of the AST. Icing users can specify which rewrites may be applied under each distinct `opt:` scope in their code or use a default set (shown in Table 1).

Table 1. Rewrites currently supported in Icing ($\circ \in \{+, *\}$)

	Name	Rewrite	Precondition
1	`fma` introduction	`x * y + z` → `fma (x,y,z)`	*application precond.*
2	$\circ$ associative	$(x \circ y) \circ z \rightarrow x \circ (y \circ z)$	*application precond.*
3	$\circ$ commutative	$x \circ y \rightarrow y \circ x$	*application precond.*
4	double negation	`- (- x)` → `x`	*x well-typed*
5	$*$ distributive	`x * (y + z)` → `(x * y) + (x * z)`	*no control dependency on optimization result*
6	`NaN` check removal	`isNaN x` → `false`	*x is not a NaN*

2.3 Semantics of Icing

Next, we explain the semantics of Icing, highlighting two distinguishing features. First, values are represented as trees instead of simple floating-point words, thus delaying evaluation of arithmetic expressions. Secondly, rewrites in the semantics are applied nondeterministically, thus relaxing floating-point evaluation enough to prove fast-math optimizations.

We define the semantics of Icing programs in Fig. 2 as a big-step judgment of the form $(cfg, E, e) \rightarrow v$. *cfg* is a configuration carrying a list of rewrites $(s \rightarrow t)$ representing allowed optimizations, and a flag tracking whether optimizations are allowed in the current program fragment under an `opt:` scope (`OptOk`). E is the (runtime) execution environment mapping free variables to values and e an Icing expression. The value v is the result of evaluating e under E using optimizations from *cfg*.

The first key idea of Icing's semantics is that expressions are not evaluated to (64-bit) floating-point words immediately; the semantics rather evaluates them into *value trees* representing their computation result. As an example, if e_1 evaluates to value tree v_1 and e_2 to v_2, the semantics returns the value tree represented as $v_1 + v_2$ instead of the result of the floating-point addition of (flattened) v_1 and v_2. The syntax of value trees is:

$$c ::= b \mid \texttt{isNaN}\, v_1 \mid v_1 \,\square\, v_2 \mid \texttt{opt:}\, c$$
$$v_1, v_2, v_3 ::= w \mid \diamond v_1 \mid v_1 \circ v_2 \mid \texttt{fma}(v_1, v_2, v_3) \mid \texttt{opt:}\, v_1$$

$$\frac{}{(cfg,\ E,\ c) \to \mathrm{c}}\ \texttt{Const} \qquad \frac{}{(cfg,\ E,\ b) \to \mathrm{b}}\ \texttt{Bool}$$

$$\frac{(cfg,\ E,\ e) \to v \quad (\diamond\, v,\ cfg)\ \mathsf{rewritesTo}\ r}{(cfg,\ E,\ \diamond\, e) \to r}\ \texttt{Unary} \qquad \frac{E\,(x) = r}{(cfg,\ E,\ x) \to r}\ \texttt{Var}$$

$$\frac{(cfg,\ E,\ e) \to vl \quad n < |vl| \quad vl[n] = r}{(cfg,\ E,\ e[n]) \to r}\ \texttt{Ith} \qquad \frac{(cfg,\ E,\ e_1) \to v_1 \quad (cfg,\ E,\ e_2) \to v_2 \quad (cfg,\ E,\ e_3) \to v_3 \quad (\texttt{fma}\ v_1\, v_2\, v_3,\ cfg)\ \mathsf{rewritesTo}\ r}{(cfg,\ E,\ \texttt{fma}\ e_1\, e_2\, e_3) \to r}\ \texttt{fma}$$

$$\frac{(cfg,\ E,\ e_1) \to v_1 \quad (cfg,\ E[x \mapsto v_1],\ e_2) \to v_2}{(cfg,\ E,\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2) \to v_2}\ \texttt{Let-bind} \qquad \frac{(cfg\ \texttt{with OptOk} := \texttt{true},\ E,\ e) \to v}{(cfg,\ E,\ \texttt{Opt}:\ e) \to v}\ \texttt{Scope}$$

$$\frac{(cfg,\ E,\ e_1) \to v_1 \quad (cfg,\ E,\ e_2) \to v_2 \quad (v_1 \circ v_2,\ cfg)\ \mathsf{rewritesTo}\ r}{(cfg,\ E,\ e_1 \circ e_2) \to r}\ \texttt{Binary} \qquad \frac{(cfg,\ E,\ c) \to cv \quad \texttt{cTree2IEEE}\, cv = \mathrm{b} \quad (cfg,\ E,\ e_b) \to r}{(cfg,\ E,\ \texttt{if}\ c\ \texttt{then}\ e_\mathrm{T}\ \texttt{else}\ e_\mathrm{F}) \to r}\ \texttt{If}$$

$$\frac{}{(cfg,\ E,\ \texttt{Map}\,(\lambda x.e)\,[]) \to []}\ \texttt{Map []} \qquad \frac{(cfg,\ E,\ s) \to v}{(cfg,\ E,\ \texttt{Fold}\,(\lambda x\, y.e)\, s\,[]) \to v}\ \texttt{Fold []}$$

$$\frac{(cfg,\ E,\ e_1) \to v_1 \quad (cfg,\ E[x \mapsto v_1],\ e) \to v_\mathrm{res} \quad (cfg,\ E,\ \texttt{Map}\,(\lambda x.e)\, el) \to vl}{(cfg,\ E,\ \texttt{Map}\,(\lambda x.e)\,(e_1 :: el)) \to v_\mathrm{res} :: vl}\ \texttt{Map cons}$$

$$\frac{(cfg,\ E,\ e_1) \to v_1 \quad (cfg,\ E,\ \texttt{Fold}\,(\lambda x\, y.e)\, s\, el) \to v_\mathrm{res} \quad (cfg,\ E[x \mapsto v_1, y \mapsto v_\mathrm{res}],\ e) \to v_\mathrm{final}}{(cfg,\ E,\ \texttt{Fold}\,(\lambda x\, y.e)\, s\,(e_1 :: el)) \to v_\mathrm{final}}\ \texttt{Fold cons}$$

$$\frac{(cfg,\ E,\ e) \to v \quad (\texttt{isNaN}\, v,\ cfg)\ \mathsf{rewritesTo}\ r}{(cfg,\ E,\ \texttt{isNaN}\, e) \to r}\ \texttt{isNaN} \qquad \frac{(cfg,\ E,\ e_1) \to v_1 \quad (cfg,\ E,\ e_2) \to v_2 \quad (v_1 \,\square\, v_2,\ cfg)\ \mathsf{rewritesTo}\ r}{(cfg,\ E,\ e_1 \,\square\, e_2) \to r}\ \texttt{Compare}$$

Fig. 2. Nondeterministic Icing semantics

```
let v1 = Map (λ x. opt:(x + 3.0)) vi in
let vsum = Fold (λ x y. opt:(x * x + y)) 0.0 v1 in sqrt vsum
```

Fig. 3. A simple Icing program

Constants are again defined as floating-point words and form the leaves of value trees (variables obtain a constant value from the execution environment E). On top of constants, value trees can represent the result of evaluating any floating-point operation Icing supports.

The second key idea of our semantics is that it nondeterministically applies rewrites from the configuration *cfg while evaluating* expression e instead of just returning its value tree. In the semantics, we model the nondeterministic choice of an optimization result for a particular value tree v with the relation `rewritesTo`, where (cfg, v) `rewritesTo` r if either the configuration *cfg* allows for optimizations to be applied, and value tree v can be rewritten into value tree r using rewrites from the configuration *cfg*; or the configuration does not allow for rewrites to be applied, and $v = r$. Rewriting on value trees reuses several definitions from Sect. 2.2. We add the nondeterminism on top of the existing functions by making the relation `rewritesTo` pick a subset of the rewrites from the configuration *cfg* which are applied to value tree v.

Icing's semantics allows optimizations to be applied for arithmetic and comparison operations. The rules `Unary`, `Binary`, `fma`, `isNaN`, and `Compare` first evaluate argument expressions into value trees. The final result is then nondeterministically chosen from the `rewritesTo` relation for the obtained value tree and the current configuration. Evaluation of `Map`, `Fold`, and let-bindings follows standard textbook evaluation semantics and does not apply optimizations.

Rule `Scope` models the fine-grained control over where optimizations are applied in the semantics. We store in the current configuration *cfg* that optimizations are allowed in the (sub-)expression e (`cfg with OptOk := true`).

Evaluation of a conditional (`if` c `then` e_T `else` e_F) first evaluates the conditional guard c to a value tree cv. Based on value tree cv the semantics picks a branch to continue evaluation in. This eager evaluation for conditionals (in contrast to delaying by leaving them in a value tree) is crucial to enable the later simulation proof to connect Icing to CakeML which also eagerly evaluates conditionals. As the value tree cv represents a delayed evaluation of a boolean value, we have to turn it into a boolean constant when selecting the branch to continue evaluation in. This is done using the functions `cTree2IEEE` and `tree2IEEE`. `cTree2IEEE (v)` computes the boolean value, and `tree2IEEE (v)` computes the floating-point word represented by the value tree `v` by applying IEEE 754 arithmetic operations and structural recursion.

Example. We illustrate Icing semantics and how optimizations are applied both in syntax and semantics with the example in Fig. 3. The example first translates the input list by 3.0 using a `Map`, and then computes the norm of the translated list with `Fold` and `sqrt`.

We want to apply $x + y \rightarrow y + x$ (commutativity of $+$) and `fma`-introduction ($x * y + z \rightarrow \texttt{fma}(x, y, z)$) to our example program. Depending on their order the function `rewrite` will produce different results.

If we first apply commutativity of $+$, and then `fma` introduction, all $+$ operations in our example will be commuted, but no `fma` introduced as the `fma` introduction *syntactically* relies on the expression having the structure $x * y + z$ where x, y, z can be arbitrary. In contrast, if we use the opposite order of rewrites, the second line will be replaced by `let vsum = Fold (λx y.fma(x,x,y)) 0.0 v1` and commutativity is only applied in the first line.

To illustrate how the semantics applies optimizations, we run the program on the 2D unit vector (`vi = [1.0,1.0]`) in a configuration that contains both rewrites. Consequently the `Map` application can produce `[1.0 + 3.0, 1.0 + 3.0]`, `[3.0 + 1.0, 1.0 + 3.0]`, ... Where the terms `1.0 + 3.0`, `3.0 + 1.0` correspond to the value trees representing the addition of `1.0` and `3.0`.

If we apply the `Fold` operation to this list, there are even more possible optimization results:

```
[(1.0 + 3.0) * (1.0 + 3.0) + (1.0 + 3.0) * (1.0 + 3.0)],
[(3.0 + 1.0) * (3.0 + 1.0) + (3.0 + 1.0) * (3.0 + 1.0)],
[fma ((3.0 + 1.0), (3.0 + 1.0), (3.0 + 1.0) * (3.0 + 1.0))],
[fma ((1.0 + 3.0), (1.0 + 3.0), (3.0 + 1.0) * (1.0 + 3.0))], ...
```

The first result is the result of evaluating the initial program without any rewrites, the second result corresponds to syntactically optimizing with commutativity of $+$ and then `fma` introduction, and the third corresponds to using the opposite order syntactically. The last two results can only be results of semantic optimizations as commutativity and `fma` introduction are applied to some intermediate results of `Map`, but not all. There is no syntactic application of commutativity and `fma`-introduction leading to such results.

3 Modelling Existing Compilers in Icing

Having defined the syntax and semantics of Icing, we next implement and prove correct functions which model the behavior of previous verified compilers, like CompCert or CakeML, and the behavior of unverified compilers, like GCC or Clang, respectively. For the former, we first define a translator of Icing expressions which preserves the IEEE 754 strict meaning of its input and does not allow for any further optimizations. Then we give a greedy optimizer that unconditionally optimizes expressions, as observed by GCC and Clang.

3.1 An IEEE 754 Preserving Translator

The Icing semantics nondeterministically applies optimizations if they are added to the configuration. However, when compiling safety-critical code or after applying some syntactic optimizations, one might want to preserve the strict IEEE 754 meaning of an expression.

To make sure that the behavior of an expression cannot be further changed and thus the expression exhibits strict IEEE 754 compliant behavior, we have implemented the function `compileIEEE754`, which essentially *disallows optimizations* by replacing all optimizable expressions `opt: e'` with non-optimizable expressions `e'`. Correctness of `compileIEEE754` shows that (a) no optimizations can be applied after the function has been applied, and (b) evaluation is deterministic. We have proven these properties as separate theorems.

3.2 A Greedy Optimizer

Next, we implement and prove correct an optimizer that mimics the (observed) behavior of GCC and Clang as closely as possible. The optimizer applies `fma` introduction, associativity and commutativity greedily. All these rewrites only have an application rewrite precondition which we instantiate to `True` to apply the rewrites unconstrained.

To give an intuition for greedy optimization, recall the example from Fig. 3. Greedy optimization does not consider whether applying an optimization is beneficial or not. If the optimization is allowed to be applied and it matches some subexpression of an optimizable expression, it is applied. Thus the order of optimizations matters. Applying the greedy optimizer with the rewrites `[associativity,fma-introduction, commutativity]` to the example, we get:

```
let v1 = Map (λ x. opt:(3.0 + x)) vi in
let vsum = Fold (λ x y. opt:(y + x * x)) 0.0 v1 in sqrt vsum
```

Only commutativity has been applied as associativity does not match and the possibility for an `fma`-introduction is ruled out by commutativity. If we reverse the list of optimizations we obtain:

```
let v1 = Map (λ x. opt:(3.0 + x)) vi in
let vsum = Fold (λ x y. opt:(fma (x,x,y))) 0.0 v1 in sqrt vsum
```

which we consider to be a more efficient version of the program from Fig. 3.

Greedy optimization is implemented in the function `optimizeGreedy (rws, e)` which applies the rewrites in `rws` in a bottom-up traversal to expression `e`. In combination with the greedy optimizer our fine-grained control (using `opt` annotations) allows the end-user to control *where* optimizations can be applied.

We have shown correctness of `optimizeGreedy` with respect to Icing semantics, i.e., we have shown that optimizing greedily gives the same result as applying the greedy rewrites in the semantics:[1]

Theorem 1. *`optimizeGreedy` is correct*
Let E be an environment, v a value tree and cfg a configuration.
If $(cfg, E, \texttt{optimizeGreedy ([associativity,commutativity,fma-intro], e)}) \rightarrow v$ *then* $(cfg \text{ with } \texttt{[associativity, commutativity,fma-intro]}, E, \texttt{e}) \rightarrow v$.

[1] As in many verified compilers, Icing's proofs closely follow the structure of optimizations. Achieving this required careful design and many iterations; we consider the simplicity of Icing's proofs to be a strength of this work.

Proving Theorem 1 without any additional lemmas is tedious as it requires showing correctness of a single optimization in the presence of other optimizations and dealing with the bottom-up traversal applying the optimization at the same time. Thus we reduce the proof of Theorem 1 to proving each rewrite separately and then chaining together these correctness proofs. Lemma 1 shows that applications of the function `rewrite` can be chained together in the semantics. This also means that adding, removing, or reordering optimizations simply requires changing the list of rewrites, thus making Icing easy to extend.

Lemma 1. *`rewrite` is compositional*
Let e be an expression, v a value tree, $s \to t$ a rewrite, and rws a set of rewrites. If the rewrite $s \to t$ can be correctly simulated in the semantics, and list rws can be correctly simulated in the semantics, then the list of rewrites $(s \to t) :: rws$ can be correctly simulated in the semantics.

4 A Conditional Optimizer

We have implemented an IEEE 754 optimizer which has the same behavior as CompCert and CakeML, and a greedy optimizer with the (observed) behavior of GCC and Clang. The fine-grained control of where optimizations are applied is essential for the usability of the greedy optimizer. However, in this section we explain that the control provided by the `opt` annotation is often not enough. We show how preconditions can be used to provide additional constraints on where rewrites can be applied, and sketch how preconditions serve as an interface between the compiler and external tools, which can and should discharge them.

We observe that in many cases, whether an optimization is acceptable or not can be captured with a precondition *on the optimization itself*, and not on every arithmetic operation separately. One example for such an optimization is removal of `NaN` checks as a check for a `NaN` should only be removed if the check never succeeds.

We argue that both application and compiler rewrite preconditions should be discharged by external tools. Many interesting preconditions for a rewrite depend on a global analysis. Running a global analysis as part of a compiler is infeasible, as maintaining separate analyses for each rewrite is not likely to scale. We thus propose to expose an *interface to external tools* in the form of preconditions.

We implement this idea in the *conditional optimizer* `optimizeCond` that supports three different applications of fast-math optimizations: applying optimizations `rws` unconstrained (`uncond rws`), applying optimizations if precondition `P` is true (`cond P rws`), and applying optimizations under the assumptions generation by function `A` which should be discharged externally (`assume A rws`). When applying `cond`, `optimizeCond` checks whether precondition `P` is true before optimizing, whereas for `assume` the propositions returned by `A` are assumed, and should then be discharged separately by a static analysis or a manual proof.

Correctness of `optimizeCond` relates syntactic optimizations to applying optimizations in the semantics. Similar to `optimizeGreedy`, we designed the proof modularly such that it suffices to prove correct each rewrite individually.

Our optimizer `optimizeCond` takes as arguments first a list of rewrite applications using `uncond`, `cond`, and `assume` then an expression `e`. If the list is empty, we have `optimizeCond ([], e) = e`. Otherwise the rewrite is applied in a bottom-up traversal to `e` and optimization continues recursively. For `uncond`, the rewrites are applied if they match; for `cond P rws` the precondition `P` is checked for the expression being optimized and the rewrites `rws` are applied if `P` is true; for `assume A rws`, the function `A` is evaluated on the expression being optimized. If execution of `A` fails, no optimization is applied. Otherwise, `A` returns a list of assumptions which are logged by the compiler and the rewrites are applied.

Using the interface provided by preconditions, one can prove external theorems showing additional properties of a compiler run using application rewrite preconditions, and external theorems showing how to discharge compiler rewrite preconditions with static analysis tools or a manual proof. We will call such external theorems *meta theorems*.

In the following we discuss two possible meta theorems, highlighting key steps required for implementing (and proving) them. A complete implementation consists of two connections: (1) from the compiler to rewrite preconditions and (2) from rewrite preconditions to external tools. We implement (1) independently of any particular tool. A complete implementation of (2) is out of scope of this paper; meta theorems generally depend on global analyses which are orthogonal to designing Icing, but several external tools already provide functionality that is a close match to our interface and we sketch possible connections below. We note that for these meta theorems, `optimizeCond` should track the context in which an assumption is made and use the context to express assumptions as *local* program properties. Our current `optimizeCond` implementation does not collect this contextual information yet, as this information at least partially depends on the particular meta theorems desired.

4.1 A Logging Compiler for NaN Special Value Checks

We show how a meta theorem can be used to discharge a compiler rewrite precondition on the example of removing a `NaN` check. Removing a `NaN` check, in general, can be unsound if the check could have succeeded. Inferring statically whether a value can be a `NaN` special value or not requires either a global static analysis, or a manual proof on all possible executions.

Preconditions are our interface to external tools. For `NaN` check removal, we implement a function `removeNaNcheck e` that returns the assumption that no `NaN` special value can be the result of evaluating the argument expression e. Function `removeNaNCheck` could then be used as part of an `assume` rule for `optimizeCond`. We prove a strengthened correctness theorem for `NaN` check removal, showing that if the assumption returned by `removeNaNcheck` is discharged externally (i.e. by the end-user or via static analysis), then we can simulate applying `NaN` check removal syntactically in Icing semantics *without additional sideconditions*.

The assumption from `removeNaNcheck` is additionally returned as the result of `optimizeCond` since it is faithfully assumed when optimizing. Such assumptions can be discharged by static analyzers like Verasco [22], or Gappa [17].

4.2 Proving Roundoff Error Improvement

Rewrites like associativity and distributivity change the results of floating-point programs. One way of capturing this behavior for a single expression is to compute the roundoff error, i.e. the difference between an idealized real-valued and a floating-point execution of the expression.

To compute an upper bound on the roundoff error, various formally verified tools have been implemented [3,17,30,37]. A possible meta theorem is thus to show that applying a particular list of optimizations does not increase the roundoff error of the optimized expression but only decreases or preserves it. The meta theorem for this example would show that (a) all the applied syntactic rewrites can be simulated in the semantics and (b) the worst-case roundoff error of the optimized expression is smaller or equal to the error of the input expression. Our development already proves (a) and we sketch the steps necessary to show (b) below.

We can leverage these roundoff error analysis tools as application preconditions in a `cond` rule, checking whether a rewrite should be applied or not in `optimizeCond`. For a particular expression `e`, an application precondition (`check (s→t, e)`) would return true if applying rewrite `s→t` does not increase the roundoff error of `e`.

Theorem 2. *`check` decreases roundoff error*
(cfg, E, `optimizeCond (Cond (λe. check (s→t, e))) e`*) → v ⟹*
(cfg with opts := cfg.opts ∪ {s → t}, E, e) → v ∧
`error e` ≤ `error (optimizeCond (Cond (λe. check (s→t, e))) e)`

Implementing `check (s→t, e)` requires computing a roundoff error for expression `e` and one for `e` rewritten with `s→t` and returning `True` if and only if the roundoff error has not increased by applying the rewrite. Proving the theorem would require giving a real-valued semantics for Icing, connecting Icing's semantics to the semantics of the roundoff error analysis tool, and a global range analysis on the Icing programs, which can be provided by Verasco or Gappa.

4.3 Supporting Distributivity in `optimizeCond`

The rewrites considered up to this point do not duplicate any subexpressions in the optimized output. In this section, we consider rewrites which do introduce additional occurrences of subexpressions, which we dub *duplicative rewrites*. Common duplicative rewrites are distributivity of $*$ with $+$ ($x * (y + x) \leftrightarrow x * y + x * z$) and rewriting a single multiplication into multiple additions ($x * n \leftrightarrow \sum_{i=1}^{n} x$). Here we consider distributivity as an example. A compiler might want to use this optimization to apply further strength reductions or `fma` introduction.

The main issue with duplicative rewrites is that they add new occurrences of a matched subexpression. Applying $(x * (y + z) \rightarrow x * y + x * z)$ to `e1 * (2 + x)` returns `e1 * 2 + e1 * x`. The values for the two occurrences of `e1` may differ because of further optimizations applied to only one of it's occurrences.

Any correctness proof for such a duplicative rewrite must match up the two (potentially different) executions of `e1` in the optimized expression (`e1 * 2 + e1 * x`) with the execution of `e1` in the initial expression (`e1 * (2 + x)`). This can only be achieved by finding a common intermediate optimization (resp. evaluation) result shared by both subexpressions of `e1 * 2 + e1 * x`.

In general, existence of such an intermediate result can only be proven for expressions that do not depend on "eager" evaluation, i.e. which consists of let-bindings and arithmetic. We illustrate the problem using a conditional (`if c then e1 else e2`). In Icing semantics, the guard `c` is first evaluated to a value tree `cv`. Next, the semantics evaluates `cv` to a boolean value b using function `cTree2IEEE`. Computing b from `cv` loses the structural information of value tree `cv` by computing the results of previously delayed arithmetic operations. This loss of information means that rewrites that previously matched the structure of `cv` may no longer apply to b.

This is not a bug in the Icing semantics. On the contrary, our semantics makes this issue explicit, while in other compilers it can lead to unexpected behavior (e.g., in GCC's support for distributivity under fast-math). CakeML, for example, also eagerly evaluates conditionals and similarly loses structural information about optimizations that otherwise may have been applied. Having lazy conditionals in general would only "postpone" the issue until eager evaluation of the conditional expression for a loop is necessary.

An intuitive compiler precondition that enables proving duplicative rewrites is to forbid any control dependencies on the expression being optimized. However, this approach may be unsatisfactory as it disallows branching on the results of optimized expressions and requires a verified dependency analysis that must be rerun or incrementally updated after every rewrite, and thus could become a bottleneck for fast-math optimizers. Instead, in Icing we restrict duplicative rewrites to only fire when pattern variables are matched against program variables, e.g., pattern variables a, b, c only match against program variables `x, y, z`. This restriction to only matching let-bound variables is more scalable, as it can easily be checked syntactically, and allows us to loosen the restriction on control-flow dependence by simply let-binding subexpressions as needed.

5 Connecting to CakeML

We have shown how to apply optimizations in Icing and how to use it to preserve IEEE 754 semantics. Next, we describe how we connected Icing to an existing verified compiler by implementing a translation from Icing source to CakeML

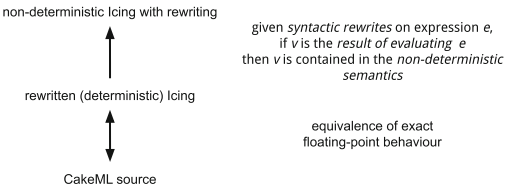

Fig. 4. Simulation diagram for Icing and the designed optimizers

source and showing an equivalence theorem.[2] The translation function `toCML` maps Icing syntax to CakeML syntax. We highlight the most interesting cases. The translations of `Ith`, `Map`, `Fold` relate an Icing execution to a predefined function from the CakeML standard library. We show separate theorems relating executions of list operations in Icing to CakeML closures of library functions. The predicate `isNaN e` is implemented as `toCML e <> toCML e`. The predicate is true in Icing semantics, if and only if `e` is a `NaN` special value. Recall that floating-point `NaN` values are incomparable (even to themselves) and thus we implement `isNaN` with an equality check.

To show that our translation function `toCML` correctly translates Icing programs into CakeML source, we proved a simulation between the two semantics, illustrated in Fig. 4. The top part consists of the correctness theorems we have shown for the optimizers, relating syntactic optimization to semantic rewriting. In the bottom part we relate a *deterministic* Icing execution which does not apply optimizations to CakeML source semantics and prove an equivalence. For the backward simulation between CakeML and Icing we require the Icing program to be well-typed which is independently checked.

6 Related Work

Verified Compilation of Floating-Point Programs. CompCert [25] uses a constructive formalization of IEEE 754 arithmetic [6] based on Flocq [7] which allows for verified constant propagation and strength reduction optimizations for divisions by powers of 2 and replacing $x \times 2$ by $x + x$. The situation is similar for CakeML [38] whose floating-point semantics is based on HOL's [19,20]. With Icing, we propose a semantics which allows important floating-point rewrites in a verified compiler by allowing users to specify a larger set of possible behaviors for their source programs. The precondition mechanism serves as an interface to external tools. While Icing is implemented in HOL, our techniques are not specific to higher-order logic or the details of CakeML and we believe that an analog of our "verified fast-math" approach could easily be ported to CompCert.

The Alive framework [27] has been extended to verify floating-point peephole optimizations [29,31]. While these tools relax some exceptional (`NaN`) cases,

[2] We also extended the CakeML source semantics with an `fma` operation, as CakeML's compilation currently does not support mapping `fma`'s to hardware instructions.

most optimizations still need to preserve "bit-for-bit" IEEE 754 behavior, which precludes valuable rewrites like the `fma` introductions Icing supports.

Optimization of Floating-Point Programs. 'Mixed-precision tuning' can increase performance by decreasing precision at the expense of accuracy, for instance from double to single floating-point precision. Current tools [11,13,16,35], ensure that a user-provided error bound is satisfied either through dynamic or static analysis. In this work, we consider only uniform 64-bit floating-point precision, but Icing's optimizations are equally applicable to other precisions. Optimizations such as mixed-precision tuning are, however, out of scope of a compiler setting, as they require error bound annotations for kernel functions.

Spiral [33] uses real-valued linear algebra identities for rewriting at the algorithmic level to choose a layout which provides the best performance for a particular platform, but due to operation reordering is not IEEE 754 semantics preserving. Herbie [32] optimizes for accuracy, and not for performance by applying rewrites which are mostly based on real-valued identities. The optimizations performed by Spiral and Herbie go beyond what traditional compilers perform, but they fit our view that it is sometimes beneficial to relax the strict IEEE 754 specification, and could be considered in an extended implementation of Icing. On the other hand, STOKE's floating-point superoptimizer [36] for x86 binaries does not preserve real-valued semantics, and only provides approximate correctness using dynamic analysis.

Analysis and Verification of Floating-Point Programs. Static analysis for bounding roundoff errors of finite-precision computations w.r.t. to a real-valued semantics [15,17,18,28,30,37] (some with formal certificates in Coq or HOL), are currently limited to short, mostly straight-line functions and require fine-grained domain annotations at the function level. Whole program accuracy can be formally verified w.r.t. to a real-valued implementation with substantial user interaction and expertise [34]. Verification of elementary function implementations has also recently been automated, but requires substantial compute resources [23].

On the other hand, static analyses aiming to verify the absence of runtime exceptions like division by zero [4,10,21,22] scale to realistic programs. We believe that such tools can be used to satisfy preconditions and thus Icing would serve as an interface between the compiler and such specialized verification techniques.

The KLEE symbolic execution engine [9] has support for floating-point programs [26] through an interface to Z3's floating-point theory [8]. This theory is also based on IEEE 754 and will thus not be able to verify the kind of optimizations that Icing supports.

7 Conclusion

We have proposed a novel semantics for IEEE 754-unsound floating-point compiler optimizations which allows them to be applied in a verified compiler setting

and which captures the intuitive semantics developers often use today when reasoning about their floating-point code. Our semantics is nondeterministic in order to provide the compiler the freedom to apply optimizations where they are useful for a particular application and platform—but within clearly defined bounds. The semantics is flexible from the developer's perspective, as it provides fine-grained control over which optimizations are available and where in a program they can be applied. We have presented a formalization in HOL4, implemented three prototype optimizers, and connected them to the CakeML verified compiler frontend. For our most general optimizer, we have explained how it can be used to obtain meta-theorems for its results by exposing a well-defined interface in the form of preconditions. We believe that our semantics can be integrated fully with different verified compilers in the future, and bridge the gap between compiler optimizations and floating-point verification techniques.

References

1. LLVM language reference manual - fast-math flags (2019). https://llvm.org/docs/LangRef.html#fast-math-flags
2. Semantics of floating point math in GCC (2019). https://gcc.gnu.org/wiki/FloatingPointMath
3. Becker, H., Zyuzin, N., Monat, R., Darulova, E., Myreen, M.O., Fox, A.: A verified certificate checker for finite-precision error bounds in Coq and HOL4. In: 2018 Formal Methods in Computer Aided Design (FMCAD), pp. 1–10. IEEE (2018)
4. Blanchet, B., et al.: A static analyzer for large safety-critical software. In: PLDI (2003)
5. Boldo, S., Jourdan, J.H., Leroy, X., Melquiond, G.: A formally-verified c compiler supporting floating-point arithmetic. In: 2013 21st IEEE Symposium on Computer Arithmetic (ARITH), pp. 107–115. IEEE (2013)
6. Boldo, S., Jourdan, J.H., Leroy, X., Melquiond, G.: Verified compilation of floating-point computations. J. Autom. Reasoning **54**(2), 135–163 (2015)
7. Boldo, S., Melquiond, G.: Flocq: a unified library for proving floating-point algorithms in Coq. In: 19th IEEE International Symposium on Computer Arithmetic, ARITH, pp. 243–252 (2011). https://doi.org/10.1109/ARITH.2011.40
8. Brain, M., Tinelli, C., Ruemmer, P., Wahl, T.: An automatable formal semantics for IEEE-754 floating-point arithmetic. Technical report (2015). http://smt-lib.org/papers/BTRW15.pdf
9. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI (2008)
10. Chen, L., Miné, A., Cousot, P.: A sound floating-point polyhedra abstract domain. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 3–18. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89330-1_2
11. Chiang, W.F., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous floating-point mixed-precision tuning. In: Symposium on Principles of Programming Languages (POPL), pp. 300–315. ACM (2017)
12. Corden, M., Kreitzer, D.: Consistency of floating-point results using the Intel compiler. Technical report, Intel Corporation (2010)
13. Damouche, N., Martel, M.: Mixed precision tuning with salsa. In: PECCS, pp. 185–194. SciTePress (2018)

14. Damouche, N., Martel, M., Chapoutot, A.: Intra-procedural optimization of the numerical accuracy of programs. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 31–46. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19458-5_3
15. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - framework for analysis and optimization of numerical programs (tool paper). In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 270–287. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_15
16. Darulova, E., Sharma, S., Horn, E.: Sound mixed-precision optimization with rewriting. In: ICCPS (2018)
17. De Dinechin, F., Lauter, C.Q., Melquiond, G.: Assisted verification of elementary functions using Gappa. In: ACM Symposium on Applied Computing, pp. 1318–1322. ACM (2006)
18. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 232–247. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_17
19. Harrison, J.: Floating point verification in HOL. In: Thomas Schubert, E., Windley, P.J., Alves-Foss, J. (eds.) TPHOLs 1995. LNCS, vol. 971, pp. 186–199. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60275-5_65
20. Harrison, J.: Floating-point verification. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 529–532. Springer, Heidelberg (2005). https://doi.org/10.1007/11526841_35
21. Jeannet, B., Miné, A.: Apron: a library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
22. Jourdan, J.H.: Verasco: a formally verified C static analyzer. Ph.D. thesis, Université Paris Diderot (Paris 7), May 2016
23. Lee, W., Sharma, R., Aiken, A.: On automatically proving the correctness of math.h implementations. In: POPL (2018)
24. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd ACM Symposium on Principles of Programming Languages, pp. 42–54. ACM Press (2006)
25. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning **43**(4), 363–446 (2009). http://xavierleroy.org/publi/compcert-backend.pdf
26. Liew, D., Schemmel, D., Cadar, C., Donaldson, A.F., Zähl, R., Wehrle, K.: Floating-point symbolic execution: a case study in n-version programming. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE Press (2017)
27. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with alive. In: PLDI (2015)
28. Magron, V., Constantinides, G., Donaldson, A.: Certified roundoff error bounds using semidefinite programming. ACM Trans. Math. Softw. **43**(4), 1–34 (2017)
29. Menendez, D., Nagarakatte, S., Gupta, A.: Alive-FP: automated verification of floating point based peephole optimizations in LLVM. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 317–337. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_16
30. Moscato, M., Titolo, L., Dutle, A., Muñoz, C.A.: Automatic estimation of verified floating-point round-off errors via static analysis. In: Tonetta, S., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP 2017. LNCS, vol. 10488, pp. 213–229. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66266-4_14

31. Nötzli, A., Brown, F.: LifeJacket: verifying precise floating-point optimizations in LLVM. In: Proceedings of the 5th ACM SIGPLAN International Workshop on State of the Art in Program Analysis, pp. 24–29. ACM (2016)
32. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. In: Conference on Programming Language Design and Implementation (PLDI) (2015)
33. Püschel, M., et al.: SPIRAL - a generator for platform-adapted libraries of signal processing alogorithms. IJHPCA **18**(1), 21–45 (2004)
34. Ramananandro, T., Mountcastle, P., Meister, B., Lethin, R.: A unified Coq framework for verifying C programs with floating-point computations. In: Certified Programs and Proofs (CPP) (2016)
35. Rubio-González, C., et al.: Precimonious: tuning assistant for floating-point precision. In: SC (2013)
36. Schkufza, E., Sharma, R., Aiken, A.: Stochastic optimization of floating-point programs with tunable precision. In: PLDI (2014)
37. Solovyev, A., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with Symbolic Taylor Expansions. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 532–550. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_33
38. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: The verified CakeML compiler backend. J. Funct. Program. **29** (2019)
39. Tristan, J.B., Leroy, X.: Formal verification of translation validators: a case study on instruction scheduling optimizations. In: Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL 2008), pp. 17–27. ACM Press, January 2008
40. Tristan, J.B., Leroy, X.: Verified validation of lazy code motion. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), pp. 316–326 (2009)
41. Tristan, J.B., Leroy, X.: A simple, verified validator for software pipelining. In: Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL 2010), pp. 83–92. ACM Press (2010)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.