# Verified Compilation and Optimization of Floating-Point Programs in CakeML

**Heiko Becker**
MPI-SWS, Saarland Informatics Campus (SIC), Saarbrücken, Germany

**Robert Rabe**
TU München, Germany

**Eva Darulova**
Uppsala University, Sweden

**Magnus O. Myreen**
Chalmers University of Technology, Gothenburg, Sweden

**Zachary Tatlock**
University of Washington, Seattle, WA, USA

**Ramana Kumar**
DeepMind, London, UK

**Yong Kiam Tan**
Carnegie Mellon University, Pittsburgh, PA, USA

**Anthony Fox**
Arm Limited, Cambridge, UK

## Abstract

Verified compilers such as CompCert and CakeML have become increasingly realistic over the last few years, but their support for floating-point arithmetic has thus far been limited. In particular, they lack the "fast-math-style" optimizations that unverified mainstream compilers perform. Supporting such optimizations in the setting of verified compilers is challenging because these optimizations, for the most part, *do not* preserve the IEEE-754 floating-point semantics. However, IEEE-754 floating-point numbers are finite approximations of the real numbers, and we argue that any compiler correctness result for fast-math optimizations should appeal to a real-valued semantics rather than the rigid IEEE-754 floating-point numbers.

This paper presents RealCake, an extension of CakeML that achieves end-to-end correctness results for fast-math-style optimized compilation of floating-point arithmetic. This result is achieved by giving CakeML a flexible floating-point semantics and integrating an external proof-producing accuracy analysis. RealCake's end-to-end theorems relate the I/O behavior of the original source program under real-number semantics to the observable I/O behavior of the compiler generated and fast-math-optimized machine code.

**2012 ACM Subject Classification** Software and its engineering → Formal software verification; Software and its engineering → Compilers; Software and its engineering → Software performance

**Keywords and phrases** compiler verification, compiler optimization, floating-point arithmetic

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2022.1

**Supplementary Material** *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.10`

**Funding** *Eva Darulova*: Part of this work was done while the author was at MPI-SWS, Germany.
*Magnus O. Myreen*: Supported by the Swedish Foundation for Strategic Research.
*Zachary Tatlock*: Supported in part by the U.S. Department of Energy under Award Number DE-SC0022081 (ComPort), and by the National Science Foundation under Grant No. 1749570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DOE or NSF.






## 1 Introduction

Verified compilers guarantee that the executable code they generate will only exhibit behaviors allowed by the semantics of the input program. Establishing such guarantees is challenging [49], especially if the compiler is to perform sophisticated optimizations. Adding new classes of optimizations requires significant design and proof engineering effort. Despite tremendous progress, state-of-the-art verified compilers like CakeML [55] for ML and CompCert [32] for C remain only moderately optimizing. While CakeML and CompCert support classic optimizations like common subexpression and dead code elimination, their compilation and optimization of floating-point programs is very limited: CompCert performs only a few conservative optimizations and, prior to this work, CakeML did not support floating-point arithmetic at all.

The limited support of floating-point programs in verified compilers is in stark contrast to mainstream compiler frameworks like GCC [19] and LLVM [4]. Both of these compiler frameworks support aggressive floating-point optimization via their `fast-math` flags [20, 57, 33, 7]. Fast-math optimizations include reassociating arithmetic, *e.g.*, rewriting $x \times (x \times (x \times x)) \rightarrow (x \times x) \times (x \times x)$ to enable common subexpression elimination; fused-multiply-add (FMA) introduction, *i.e.*, rewriting $x \times y + z \rightarrow \texttt{fma}(x, y, z)$ for strength reduction and to avoid intermediate rounding; as well as branch folding and dead code elimination by assuming special floating-point values like Not-a-Number (NaN) do not arise.

While such optimizations are sound under real-number semantics, they generally do *not* preserve IEEE-754 behavior [26], *e.g.*, because floating-point arithmetic is non-associative due to the inherent rounding at every intermediate operation. In part, this is why verified compilers up until now have not supported fast-math optimizations: CompCert strictly preserves IEEE-754 semantics [10], under which such optimizations are disallowed.

However, for many applications strict preservation of IEEE-754 semantics is overly constraining and artificial, preventing useful performance optimizations. Numerical applications are typically *designed* implicitly assuming real-number arithmetic and are only later *implemented* in floating-point arithmetic.[1]

The Icing language [5] proposes a more relaxed, non-deterministic semantics for floating-point expressions that allows a limited set of fast-math-style optimizations to be applied. Icing comes with a proof-of-concept optimizer, whose formal correctness proof shows that the optimization result is one of those modeled by the semantics of the initial expression.

While Icing showed what it means to allow fast-math-style optimizations in a verified compiler, Icing did not go as far as bounding the accuracy of the resulting, fast-math-optimized code. Icing's correctness theorems describe only the optimizations that a verified compiler can apply to a floating-point expression, but not their effect on overall program behavior. Hence the Icing optimizer cannot bound changes in the accuracy of the optimized floating-point expression with respect to the real-valued semantics of the unoptimized expression.

We argue that a verified compiler must provide accuracy guarantees to reasonably support fast-math-style optimizations. Applications in domains such as signal processing [12], embedded controllers [38], and neural networks [23], which could be optimized with fast-math-style optimizations, are designed to operate in noisy environments and can thus tolerate a certain amount of floating-point roundoff error by design, however, this noise has to be *bounded*. For example, a real-number version of an embedded controller is typically proven correct (i.e. stable) with respect to *bounded* implementation noise [3]. At the same time, performance is important and so developers are often indifferent to fine-grained floating-point implementation decisions.

[1] Error-free computation with rational or constructive real [8] libraries is often prohibitively expensive.

To support such potentially safety-critical applications in a verified compiler, we introduce a local, more flexible notion of correctness which we call *error refinement*: a floating-point kernel within an application may be optimized (potentially changing its IEEE-754 behavior) as long as its results remain within a user-specified error bound relative to the implicit real-number semantics.

We formalize error refinement inside the CakeML compiler to support fast-math optimizations *end-to-end*. Our extension, which we call *RealCake*, carries source-level guarantees down to fast-math-optimized executable machine code. That is, our final correctness theorem shows that running the *machine code* for a fast-math-optimized floating-point program under strict IEEE-754 semantics produces a result that is within a programmer-provided error bound w.r.t. the *unoptimized program* evaluated under real-number semantics. While our extension is done in the context of the CakeML compiler, we expect it to carry over to other verified compilers like CompCert as well.

Our first key technical contribution is a relaxed floating-point semantics that allows both fast-math-style optimizations as well as *backward simulation* soundness proofs (as CakeML's semantics requires determinism). RealCake's relaxed semantics preserves the core ideas of the existing Icing semantics [5] and models nondeterministic application of an arbitrary number of fast-math rewrites, just as Icing does. Icing provides only a loose coupling with CakeML via a simulation between the deterministic optimized expression and a CakeML source program. Unlike Icing, RealCake's semantics is designed to be more tightly integrated with the CakeML source semantics. This new integration is necessary to prove end-to-end error refinement that relates *unoptimized real-valued* CakeML programs and *optimized floating-point machine code*. RealCake's design furthermore supports function calls, I/O and memory beyond (Icing supported) floating-point expressions and can thus prove error refinement for complete applications.

The second technical contribution is to realize error refinement with *translation validation* [45, 51] using an interface to an existing proof-producing roundoff error bound analysis [6]. RealCake automatically composes the error bound proofs with its optimizer's correctness theorems to support fast-math optimizations with semantic and accuracy guarantees within a verified compiler for the first time.

RealCake is primarily designed to support *numerical kernels*, straight-line code such as those found in (safety-critical) embedded controllers or sensor-processing applications.[2] Often such kernels are evaluated in a control loop or process sensor inputs repeatedly. For such programs, both correctness as well as performance are important, and an analysis of the straight-line code is sufficient: the correctness (stability) of the overall programs (and loops) can be shown with, *e.g.*, control-theoretic techniques that rely on the straight-line loop body's errors being bounded [3, 37]. We do not address some of the orthogonal challenges in bounding the floating-point roundoff error for programs with loops and conditional statements, which remain open research problems [15]; state-of-the-art proof-producing error bound analyses only robustly support straight-line numerical kernels [54, 41, 6]. A key aspect of RealCake's design is the loose coupling between the compiler and error analysis. This loose coupling leads to a clean separation of concerns, which we hope will allow us to switch to more general error analysis methods when such are discovered.

We evaluated RealCake by optimizing all kernels from the standard floating-point arithmetic benchmark suite FPBench [14] (Section 7) which can be expressed as input to RealCake, for a total of 51 kernels. During our evaluation we found that CakeML was missing a general

[2] RealCake nevertheless proves error refinement for whole programs, including I/O (Section 7).

```
1 (* target error bound: 2^-5,
     precondition P:
3    0.0 ≤ x1 ≤ 5.0 ∧ −20.0 ≤ x2 ≤ 5.0 *)
  fun jetEngine(x1:double, x2:double):double =
5  opt: (let
   val t = (((3.0 * x1) * x1) + (2.0 * x2)) - x1
7  val t2 = (((3.0 * x1) * x1) - (2.0 * x2))
     - x1
9  val d = (x1 * x1) + 1.0
   val s = t / d
11 val s2 = t2 / d
   in
13 x1 + (((((((((2.0 * x1) * s) * (s - 3.0)) +
   ((x1 * x1) * ((4.0 * s) - 6.0))) * d) +
15 (((3.0 * x1) * x1) * s)) +
   ((x1 * x1) * x1)) + x1) + (3.0 * s2))
17 end)
```

**(a)** Unoptimized floating-point kernel.

```
1 (* guaranteed error bound: 2^-5,
     precondition P:
3    0.0 ≤ x1 ≤ 5.0 ∧ −20.0 ≤ x2 ≤ 5.0 *)
  fun jetEngine(x1:double, x2:double):double =
5  noopt: (let
   val t = fma((x1+x1)+x1, x1, (x2 + x2) - x1)
7  val t2 = fma((x1+x1)+x1, x1,
     fma(-2.0, x2, -x1))
9  val d = fma(x1, x1, 1.0)
   val s = t / d
11 val s2 = t2 / d
   in
13 x1 + fma(x1 * d, fma((s - 3.0) + (s - 3.0),
     s, x1 * fma(4.0, s, -6.0)),
15   fma(x1 * x1, ((s + s) + s) + x1,
     x1 + ((s2 + s2) + s2)))
17 end)
```

**(b)** Optimized floating-point kernel.

```
1 fun main () = let
    val args = Commandline.arguments ()
3   val a = Double.fromString (List.nth args 1)
    val b = Double.fromString (List.nth args 2)
5   val r = jetEngine (a, b)
    in
7     TextIO.print (Double.toString r)
    end
```

**(c)** Main function.

**Figure 1** Example unoptimized and optimized CakeML floating-point kernels, and a stand-in main function. The `opt:` annotation (lines 5) allows developers to selectively apply optimizations. Here, we choose to optimize the entire kernel, but a user may place only part of a program under `opt:` and the rest will be compiled preserving IEEE-754 semantics.

optimization that is particularly effective for floating-point programs: global constant lifting. RealCake achieves a (geometric) mean performance improvement for fast-math optimizations of 3% and a maximum improvement of 16% on top of improvements from constant lifting with respect to the unoptimized FPBench kernels. Our additional constant lifting optimization achieves a geometric mean performance improvement of 83% across all benchmarks with speedups of up-to 97%. For all optimized kernels, RealCake formally guarantees that the roundoff error remains within a user-specified bound.

## Contributions

To summarize, this paper makes the following contributions:

- the concept of error refinement and its formalization within the CakeML verified compiler (Section 2);
- an extension of CakeML with strict, IEEE-754 semantics preserving, floating-point arithmetic as well as a relaxed non-deterministic floating-point semantics (Section 4);
- a fast-math optimizer that is effective in improving the performance of floating-point programs (Section 5);
- automated proof tools that soundly bound roundoff errors of (optimized or unoptimized) kernels w.r.t. our new real-number semantics for CakeML (Section 6).

The RealCake development is publicly available.

# 2 Overview

We start by demonstrating at a high-level how RealCake works using an example, before giving an overview of the RealCake toolchain and our major design decisions.

## 2.1 Example

Figure 1a shows `jetEngine`, a straight-line nonlinear embedded controller [3] adapted to CakeML syntax. This controller has been proven to be safe for the dynamical system of a jet engine compressor. That is, it has been proven that the two state variables ($x_1$, $x_2$) will remain within the bounds given by $P$ (on line 2), that the system will always steer the state variables towards the equilibrium point (0.0), and that the systems thus remains stable. The stability proof assumes the control expression to be real-valued, but accounts for a certain amount of error, including measurement and implementation errors, and hence the controller is stable as long as the errors remain below this bound. For the purpose of this paper, we choose $2^{-5}$ as the bound on the roundoff error due to a floating-point implementation. However, in addition to stability, performance is also an important concern when designing controllers for resource-constrained embedded systems. To summarize, an embedded developer designs a controller, such as the `jetEngine`, assuming real-valued arithmetic together with an error bound, and requires that the executed finite-precision code A) correctly implements the control expression, B) is as efficient as possible, and C) meets the error bound.

Such a kernel may be part of a safety critical system so we would like to compile it to an executable using a verified compiler. Unfortunately, no verified compiler today meets the requirements listed above: while CompCert [10] does support floating-point arithmetic, and so ensures A), it does not optimize floating-point programs and cannot provide roundoff error bounds. Prior to our work, CakeML did not even support floating-point arithmetic.

RealCake, our extension of the CakeML compiler, closes this gap. RealCake automatically optimizes the input kernel into the optimized version shown in Figure 1b, compiles it down to machine code, and proves the end-to-end correctness theorem shown in Figure 2 that captures both "traditional" compiler correctness as well as accuracy guarantees.

For this example, RealCake prepares the program for optimization by replacing floating-point subtraction by addition of the inverse ($(4.0 \times s) - 6.0 \rightarrow (4.0 \times s) + (-1 \times 6.0)$), and during optimization, RealCake replaces multiplications by additions ($2.0 \times x1 \rightarrow x1 + x1$), and introduces FMA instructions ($x1 \times x1 + 1.0 \rightarrow \texttt{fma}(x1, x1, 1.0)$) that go beyond IEEE-754 semantics. For this example, RealCake compiles the optimized floating-point kernel (Figure 1b) together with a simple stand-in main function (Figure 1c) into a verified binary. On a Raspberry Pi v3, RealCake improves the performance of our example kernel by 95%. This performance improvement comes from both floating-point specific optimizations, as well as global constant lifting that is not specific but particularly effective for floating-points and that CakeML did not support before (Section 7). Such a speedup is important for repeatedly run code such as our embedded controller.

RealCake automatically proves the end-to-end correctness theorem that we formally state in Figure 2. At a high-level, Theorem 1 relates the behavior of the optimized program with the behavior of the real-number semantics of the initial, unoptimized program on the domain specified by precondition $P$.[3]

[3] The precondition is important, since roundoff errors directly depend on the ranges of (intermediate) values.

▶ **Theorem 1** (*jetEngine* – Whole program correctness).

$$\texttt{jetEngineInputsInPrecond}\ (s_1, s_2)\ (w_1, w_2)\ \wedge\ \texttt{environmentOk}\ ([\mathit{jetEngine};\ s_1;\ s_2], \mathit{fs})\ \Rightarrow$$
$$\exists\, w\ r.$$
$$\quad \texttt{CakeMLevaluatesAndPrints}\ (\mathit{jetEngineCode}, s_1, s_2, \mathit{fs})\ (\texttt{toString}\ w)\ \wedge$$
$$\quad \texttt{initialFPcodeReturns}\ \mathit{jetEngineUnopt}\ (w_1, w_2)\ w\ \wedge$$
$$\quad \texttt{realSemanticsReturns}\ \mathit{jetEngineUnopt}\ (w_1, w_2)\ r\ \wedge\ \texttt{abs}\ (\texttt{fpToReal}\ w\ -\ r)\ \leq\ 2^{-5}$$

**Figure 2** RealCake-proven specification theorem for the `jetEngine` kernel. Here, *jetEngineCode* refers to the overall program consisting of the *jetEngine* kernel, the *main* function from Figure 1c, and the glue-code for I/O, *jetEngine* is the name of the produced binary, and *jetEngineUnopt* is the kernel from Figure 1a.

Formally, the theorem states that, if the kernel is run on two arbitrary input strings $s_1$ and $s_2$, representing the double word inputs $w_1$ and $w_2$ respectively, and the double words satisfy precondition $P$ from Figure 1a (assumption `jetEngineInputsInPrecond`), and if the machine code for the `jetEngine` kernel is run with three command line arguments (the name of the binary, and $s_1$ and $s_2$) in an environment with filesystem `fs` (assumption `environmentOk`), then there exists a double-precision floating-point word $w$ such that

**(a)** running the optimized kernel with the command line arguments prints the word $w$ on stdout[4]

**(b)** $w$ is a result of running the unoptimized `jetEngine` kernel on $(w_1, w_2)$ with optimizations applied by our relaxed semantics, and

**(c)** running the *initial unoptimized* `jetEngine` kernel under real-number semantics on $(w_1, w_2)$ returns a real number $r$ such that $|w - r| \leq 2^{-5}$, where $2^{-5}$ is the user-given error bound.

## 2.2 Overview of CakeML

RealCake extends the CakeML compiler toolchain [55], built around a verified compiler for (a dialect of) Standard ML (SML). CakeML compiles programs written in SML to x86, ARMv7, ARMv8, MIPS, RISC-V and Silver [35] machine code and is implemented completely in the HOL4 theorem prover [53]. Our work mainly focuses on the compiler part of the CakeML ecosystem.

The behavior of a program written in the CakeML dialect of SML is defined in the CakeML source semantics. This semantics is implemented as a deterministic function in the HOL4 theorem prover, in the style of functional big-step semantics [43]. CakeML programs are turned into machine code using the in-logic compiler, with compiler passes going through various intermediate languages.

The CakeML compiler's correctness theorem states that the compiler preserves observable behaviors of the input program, modulo out-of-memory errors that can occur in the generated machine code [21].

[4] We chose printing to standard output as one option for implementing I/O behavior to show how the error bound proof can be related to I/O behavior. In a real-world setting this could be replaced by other I/O functionality.

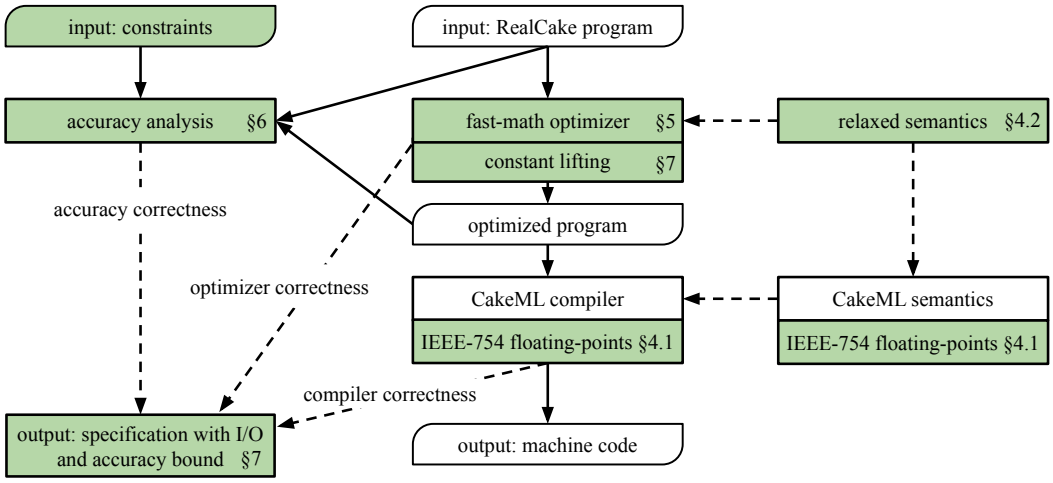


**Figure 3** Overview of the RealCake toolchain. Boxes with white background are part of the original CakeML toolchain, our RealCake extensions are marked with a green background, dashed lines indicate proof dependencies, solid lines indicate output flows.

## 2.3 Overview of RealCake

Figure 3 illustrates the RealCake toolchain, with the extensions over CakeML marked in green. The RealCake toolchain takes as inputs a program and constraints similar to those in Figure 1a. In a first step, the fast-math optimizer is run on each floating-point kernel, optimizing it with respect to RealCake's relaxed floating-point semantics, as well as lifting constants. As a result we obtain an optimized floating-point kernel, and a proof relating executions of the optimized kernel back to its unoptimized version. Next, the input constraints, and the optimized kernel are run through our accuracy analysis pipeline (left part of Figure 3). We have proven once and for all that if the analysis succeeds, the roundoff error of the optimized floating-point kernel with respect to a real-number semantics of its unoptimized version is below the user specified error bound. This requires non-trivially combining properties of the fast-math optimizer with a simulation proof relating results of the roundoff error analysis to CakeML floating-point programs. Finally, the CakeML compiler compiles the optimized kernel into machine code that can be run on x86-64 and ARMv7 platforms.[5] RealCake automatically combines optimizer correctness with the correctness of the accuracy analysis and the CakeML compiler correctness theorem to prove a theorem about the *I/O behavior* and the *accuracy* of the machine code with respect to the real-number semantics of the unoptimized, initial kernel.

One of our key insights is to apply the fast-math optimizations that require reasoning about nondeterministic semantics early; a nondeterministic semantics can be integrated more easily with a deterministic verified compiler by resolving the nondeterminism before the code enters the compiler itself. We formally prove the correctness of the fast-math optimizer: if the optimizer turns kernel $p_1$ into kernel $p_2$, and evaluating $p_2$ returns the floating-point word $w$, then the relaxed floating-point semantics can evaluate and optimize $p_1$ such that it returns $w$.

[5] At the time of writing, only the underlying ISA models for x86-64 and ARMv7 support floating-point arithmetic in CakeML.

## 2.4 Error Refinement

Optimization correctness alone effectively captures only the machine's point of view, and ignores the programmer's (implicit) real-valued source semantics. To relate the real-number, unoptimized program with its fast-math-optimized version requires both proving the correctness of our optimizer (*i.e.* showing that the behavior of the source semantics is preserved), as well as establishing accuracy guarantees using roundoff errors. Any fast-math optimization will necessarily change the rounding and thus the result value of the floating-point kernel, ruling out alternative bit-wise comparisons. While the programmer will be indifferent to how exactly the floating-point code is compiled and will accept some roundoff error – or she would not have chosen finite-precision arithmetic – this roundoff error should not be unduly large and make the computed results useless.[6] We argue that a correctness theorem of verified fast-math floating-point compilation thus needs to capture this error refinement.

To this end, RealCake automatically infers verified accuracy bounds via a verified translation from CakeML source to the proof-producing formally verified static analysis tool FloVer [6]. We combine a simulation proof relating the floating-point semantics of FloVer and CakeML with a proof that all optimizations done by our fast-math optimizer are real-valued identities. RealCake then lifts the roundoff error bound to the complete program and combines it with the general compilation correctness proofs to automatically show the end-to-end correctness theorem for our example (Figure 2). This makes RealCake the first verified compiler for floating-point arithmetic that proves a whole-program specification relating the I/O behavior of optimized floating-point machine code to the real-number semantics of the unoptimized initial program.

We choose to integrate the roundoff error analysis only loosely with CakeML. This gives us a flexible compiler infrastructure that allows us to prove roundoff error bounds on optimized as well as unoptimized floating-point kernels, or to greedily optimize kernels without necessarily proving roundoff error bounds (but still obtaining compiler correctness guarantees). By not tightly integrating the roundoff error analysis into CakeML, we have the option to relatively easily replace FloVer with an extension or another tool in the future.

RealCake's end-to-end correctness theorem only relies on error bounds proven independently for each straight-line kernel instead of a global kernel error bound. Our focus on straight-line kernels is inherited from the current capabilities of verified floating-point error analyses (see Subsection 3.2 for a more detailed discussion), but can be easily lifted with advances in this area. Per-kernel error analysis, on the other hand, is crucial to maintaining compiler modularity: it is not (nor should it be) the compiler's responsibility to ensure that a program is globally numerically stable – that is a job for the algorithm designer. Rather, the compiler compiles *and optimizes* a program and, in the case of fast-math floating-point optimizations, ensures that it has preserved sufficient (user-provided) accuracy bounds with respect to a specification over real-number semantics. This can be checked locally.

Similarly, the goal of the accuracy analysis is not necessarily to improve the accuracy of a given kernel, even though introducing FMAs will generally have this effect, but rather to ensure that the compiler has not introduced unacceptable numerical instability by accident.

Overall, a key challenge of RealCake is proof engineering. RealCake combines a verified roundoff error analysis with the deterministic CakeML compiler and a non-deterministic semantics that supports floating-point optimizations. Specifically, the main proof engineering

[6] There are programs, such as compensated sum algorithms [25], that explicitly rely on the exact floating-point semantics; such code would not be subject to fast-math-style optimizations and thus not written under an `opt:` scope.

challenge is getting the different tools to "cooperate". CakeML's source semantics is an integral part of the CakeML ecosystem. Therefore, our integration of the relaxed floating-point semantics must make sure to not break any existing invariants. Further, the semantics of the external roundoff error analysis and the semantics of CakeML source programs must be compatible such that analysis results can be transformed into CakeML source properties. Finally, all of this has to happen while making sure that RealCake optimizes floating-point programs with a non-deterministic relaxed floating-point semantics.

# 3 Background

In this section, we review some necessary background on IEEE-754 floating-point arithmetic, how to analyze the roundoff error of IEEE-754 floating-point programs, and how the Icing semantics allows to support fast-math-style optimizations in a verified compiler context.

## 3.1 IEEE-754 Floating-Point Arithmetic

The IEEE-754 standard [26] defines the representation, special values, arithmetic operations, and rounding modes of floating-point arithmetic. A floating-point number $x$ is represented as a triple $(s, m, e)$ that defines $x = (-1)^s \times m \times 2^e$ where $s$ is the sign bit, $m$ the so-called significand and $e$ the exponent. Most commonly used formats are binary single `float` and `double` precision that use 23 and 52 bits for the significand and 8 and 11 bits for the exponent, respectively. If the exponent of a number is 0, it is called a subnormal number, all other valid numerical values are called normal numbers. IEEE-754 additionally defines the special values *Infinity* and *NaN* that represent exceptional results. The standard further specifies that arithmetic operations (*e.g.*, $+, -, \times, /$) are rounded correctly, i.e. the result is as if the computation was performed in infinite precision and then rounded. The standard defines five rounding modes, of which we assume and support the most commonly used: rounding to nearest, ties to even. A further consequence of the finite precision and rounding is that floating-point arithmetic does not satisfy common real-valued identities such as associativity and distributivity. Hence, reordering a computation may lead to different results (and roundoff errors), even though the expression is equivalent under the reals.

## 3.2 Analysis of Rounding Errors

There exist a number of analysis tools that bound absolute roundoff errors for floating-point kernels and that provide formally verified error bounds: Precisa [56], FPTaylor [54], real2Float [36], Gappa [17], and FloVer [6]. They either use a global optimization analysis approach, or a forward dataflow analysis using an interval abstract domain to compute absolute roundoff errors:

$$\max_{x \in P(x)} |f(x) - \tilde{f}(x)| \tag{1}$$

where $f$ is the real-number expression, $\tilde{f}$ its floating-point counterpart and $P(x)$ is the precondition that constrains the input variables $x$. A precondition providing lower and upper bounds on the inputs is necessary to obtain interesting, non-infinite roundoff error bounds. Computing relative errors $|f(x) - \tilde{f}(x)|/|f(x)|$ is not well-defined when the denominator is zero and is thus not suitable for a general error analysis. For our purpose of checking that compilation has not introduced large numerical instabilities, any of the above mentioned tools is in principle suitable. We choose FloVer, because it is conveniently implemented in HOL4.

Despite the abundance of analysis tools, bounding finite-precision roundoff errors remains a challenging and active research area. We thus choose to focus on verifying absolute roundoff errors w.r.t. a real-number specification for straight-line arithmetic kernels, which is currently well supported. To the best of our knowledge, all available verified roundoff error analysis tools relate roundoff errors to idealized real-number semantics. Support for conditionals and loops [41] is currently severely limited. The challenge with loops is that roundoff errors in general grow in every loop iteration and thus computed fixpoints necessarily become infinite. We thus focus on absolute error accuracy analysis of straight-line arithmetic kernels, i.e. binary arithmetic operations $(+, -, \times, /)$, unary $-$, `fma` operations and let-bindings.

FloVer is a verified certificate checker for finite-precision roundoff errors that is meant to validate results of external, unverified, floating-point analysis tools. Given a certificate, encoding the result of the external analysis tool, FloVer verifies the bounds encoded in the certificate by computing the bounds using a dataflow analysis in logic. In this work in RealCake, we extend FloVer with an unverified function that computes a roundoff error certificate, which we then send through the checking pipeline.

FloVer abstracts floating-point arithmetic operations by:

$$(x \circ_{fl} y) = (x \circ y)(1 + e) \quad |e| \leq \varepsilon \tag{2}$$

where $\circ \in \{+, -, \times, /\}$ and $\varepsilon$ is the machine epsilon. FloVer uses interval arithmetic [40] to propagate intermediate bounds on $x$ and $y$, on which the magnitude of absolute errors and error propagation depends, and equally uses interval arithmetic to propagate worst-case error bounds through the arithmetic expression. For our evaluation, we have added support for `sqrt` operations to FloVer which was previously unsupported.

FloVer's soundness theorem states that a successful run of FloVer implies that the encoded floating-point kernel can be run under IEEE-754 floating-point semantics, and that the given error bound is a sound upper bound on the worst-case absolute error between the floating-point execution and the idealized real-number semantics. Error bounds proven by FloVer in RealCake are valid for normal and subnormal floating point numbers. To make this possible we extended FloVer with support for subnormal floating-point numbers, and have proven the corresponding HOL4 theorems.

## 3.3 Icing Floating-Point Semantics

Icing [5] was proposed as the first semantics to support fast-math-style optimizations in a verified compiler, going beyond IEEE-754 floating-point semantics. To support fast-math-style optimizations, Icing relies on three core ideas: fine-grained control, giving the programmer full control over which part of the program is optimized; value trees, which are a lazy representation of floating-point values; and nondeterministic floating-point evaluation, applying optimizations while evaluating. We review the design rationale behind each of these points.

**Fine-Grained Control** In unverified compilers fast-math optimizations are globally switched on or off. For a verified compiler this is unsatisfactory, as some code can be heavily optimized, while some of it might need to be compiled under strict IEEE-754 semantics. Icing solved this issue by introducing fine-grained control over which part of a program is optimized by annotating it with `opt:`, if optimizations should be applied, and `noopt:`, if optimizations should not be applied.

**Value Trees** In Icing, floating-point values are not represented by 64-bit words. Instead, Icing uses a lazy datatype called value trees. A value tree is a tree with constants as leaf nodes, and operators as intermediate nodes. During evaluation, expression variables


**Figure 4** Value trees for the unoptimized example expression (left) and its optimized version with an FMA instruction (right).

are replaced by a value tree loaded from an execution environment. When evaluating a floating-point comparison, value trees are eagerly compressed into floating-point words. Value trees are a perfect fit for encoding syntactic information about the evaluated expression in the Icing semantics.

**Nondeterministic Semantics** To handle fast-math-style optimizations in the semantics, Icing adds a set of allowed optimizations to the semantics, and the semantics can nondeterministically optimize by applying a subset of the allowed optimizations to value trees.
Icing's lazy value trees allow the semantics to alter the structure of a floating-point value after it has been evaluated. This is key to modelling the floating-point optimizations.

## Example

To illustrate how the Icing semantics works and how optimizations are applied we give a simple example. We will optimize the floating-point expression `opt:(x * 3.5 + 9.1)` in the Icing semantics.

If we evaluate the initial, unoptimized expression (`opt:(x * 3.5 + 9.1)`) in Icing semantics, which nondeterministically optimizes by introducing FMA instructions, the semantics first computes the value tree at the left-hand side of Figure 4. Because of the nondeterminism, the semantics can now either keep the value tree as is, or introduce an FMA, replacing the value tree by the one on the right-hand side of Figure 4. We explain next how Icing establishes a relation between nondeterministic optimizations and evaluation of optimized expressions to prove correctness of optimizers.

## Correctness Proofs

The original Icing paper presents three different optimizers. Here, we focus on the so-called *greedy optimizer*, and the *IEEE-754-translator*. For a list of optimizations, the greedy optimizer greedily applies them to a program wherever possible. The IEEE-754-translator rules out further optimizations by replacing all `opt:` optimization annotations by `noopt:`.

Correctness of the IEEE-754-translator proves that program evaluation is deterministic after applying the IEEE-754-translator, as no optimizations can be applied syntactically or semantically. The main correctness theorem for the greedy optimizer states: Suppose the greedy optimizer is run with optimizations $o$ on floating-point program $f$ and returns program $g$. If evaluating $g$ without any optimizations under Icing semantics gives value tree $v$, then one of the results of evaluating $f$ nondeterministically under Icing semantics with optimizations $o$ enabled is also value tree $v$. In general we call such a proof a *backwards simulation*, as it relates the result obtained from evaluating the optimized program back to an evaluation of the initial program with the applied optimizations added to the semantics.

While being an important first step to support fast-math-style optimizations in a verified compiler, Icing is not able to prove accuracy guarantees, which are required to prove end-to-end error refinement. Even though the original Icing paper proposes to include roundoff

error bounds in future work ([5], Theorem 1), we found that these guarantees could not be translated into guarantees for CakeML programs, as the backwards simulation only allows transferring information from CakeML programs to Icing expressions, but not vice versa. This motivates our approach of tightly integrating a relaxed floating-point semantics with CakeML source semantics. It allows to establish accuracy guarantees and carry them down to machine code generated by the compiler.

# 4 RealCake's Semantics

Our overall goal for RealCake is to optimize and compile floating-point kernels, establishing verified end-to-end correctness and accuracy guarantees. In this section, we lay the foundations for this work by extending the CakeML compiler with three different semantics: strict IEEE-754 preserving floating-point arithmetic, relaxed floating-point semantics going beyond IEEE-754, and a real-number semantics as a ground truth for bounding errors.

## 4.1 Extending CakeML with IEEE-754 Floating-Point Arithmetic

Prior to this paper, CakeML did not have support for floating-point arithmetic. In a first step, we add strictly IEEE-754 compliant floating-point arithmetic to CakeML. This part of the work did not require any deep new insights; we briefly review the supported operations.

The CakeML source language already had support for 64-bit machine words and we used these to hold IEEE-754 double values, but added new primitive operations for 64-bit words (single precision floats are currently not supported). The new source-level primitive operations are floating-point addition, subtraction, multiplication, division; multiply-and-add, negation, square root, absolute value, and the usual floating-point comparisons. The semantics was defined using an existing formalization of IEEE-754 by Harrison [24], which includes NaNs and Infinities.[7]

The bulk of the compiler required only simple changes since most intermediate languages compile the strict IEEE-754 operations to their very similar counterparts in the next intermediate language. The only internal part that required a bit more effort is the point where the data abstraction is implemented, i.e. where all data becomes concrete. At this point we had to wrap every primitive floating-point operation with code that unboxes and then boxes the double values. The same code is also responsible for loading and storing to the architecture-specific floating-point registers. Our IEEE-754-preserving compilation ensures that evaluation order is preserved.

At the time of writing, the CakeML compiler has six target languages. We have added floating-point support to two of them: the x86-64 and ARMv7 targets. The ARMv7 model that we use already included IEEE-754 floating-point support based on the same standard formalization that the CakeML semantics uses. For x86-64, we extended the model of the x86-64 instruction set architecture to include a minimal collection of IEEE-754 floating-point instructions required by the compiler. As the underlying L3 model [18] of the x86-64 instruction set architecture does not support FMA instructions, only the ARMv7 backend currently supports code generation for kernels with FMA instructions.

[7] The fragment of IEEE-754 that we include in CakeML has not changed between standard revisions.

```
evaluate st env [App op es] =
case evaluate st env es of
(st', Rerr v) => (st', Rerr v)
(st', Rval vs) =>
  case do_app (st'.refs, st'.ffi) op vs of
  None => (st', Rerr (Rabort Rtype_error))
  | Some ((refs, ffi), r) =>
    (updateState st' refs ffi, list_result r)
```

**(a)** Standard operator evaluation.

```
evaluate st env [FpOptimise ann e] =
case evaluate (updateOptFlag st ann) env [e] of
(st', Rerr e) => (resetOptFlag st' st, Rerr e)
| (st', Rval vs) =>
  (resetOptFlag st' st, Rval (addAnnot ann vs))
```

**(b)** Optimization scope evaluation.

```
evaluate st env [App op es] =
case evaluate st env es of
(st', Rerr v) => (st', Rerr v)
| (st', Rval vs) =>
  case do_app (st'.refs, st'.ffi) op vs of
  None => (st', Rerr (Rabort Rtype_error))
  | Some ((refs, ffi), r) =>
  let (st', r_opt) = optimizeIfOk st' r
     fp_res = if isFpBool op then toBool r_opt
              else r_opt
  in
  (updateState st' refs ffi, list_result r)
```

**(c)** Relaxed floating-point evaluation.

```
evaluate st env [App op es] =
case evaluate st env es of
(st', Rerr v) => (st', Rerr v)
| (st', Rval vs) =>
if ¬realsAllowed st'.fp_state then
  (advanceOracle st',
   Rerr (Rabort Rtype_error))
else
  case do_app (st'.refs, st'.ffi) op vs of
  None => (st', Rerr (Rabort Rtype_error))
  | Some ((refs, ffi), r) =>
    (updateState st' refs ffi, list_result r)
```

**(d)** Real-valued operator evaluation.

**Figure 5** HOL4 definitions of operator evaluation in CakeML source for (a) the simple case, (b) relaxed floating-point operations, (c) optimization scopes, and (d) real-number operations. In (c) and (d) difference to Figure 5a is highlighted in bold.

## 4.2 RealCake's Relaxed Floating-Point Semantics

Next, we present RealCake's relaxed floating-point semantics. Similar to Icing, the relaxed floating-point semantics applies optimizations during evaluation. In CakeML, we call the process of applying optimizations to floating-point kernels during evaluation *semantic optimization*. Before going into the details of how evaluation is implemented in the relaxed semantics, we briefly review some necessary details of CakeML's source semantics.

### CakeML Source Semantics

The CakeML source semantics is implemented in the style of functional big-step semantics [43]. As such, CakeML source semantics (`evaluate`) is a pure, deterministic function in the HOL4 theorem prover. `evaluate` $st_1$ *env* $e$ = $(st_2, r)$ means that evaluating the CakeML source expression $e$ under environment *env* and global state $st_1$ results in global state $st_2$ and ends with result $r$. If evaluation succeeded, $r$ is a value, otherwise $r$ is an error. Global state $st_1$ and $st_2$ model the state of the foreign-function-interface (FFI), as well as global references. In CakeML source semantics, the FFI models interactions with the outside world, *e.g.* I/O.

We explain the case for operator evaluation in more detail, as we will extend it with relaxed floating-point operations later. The definition of operator evaluation in CakeML is given in Figure 5a. In CakeML source, an operator application is written as `App` *op es*, denoting that operator *op* is applied to the list of expressions *es*.

First, `evaluate` is run on the argument list. If evaluation of the argument list fails with an error and a new state, the error and the state are returned. If evaluation succeeds returning values *vs*, function `do_app` applies operator *op* to the value list *vs* for the current references

($st'$.refs), and the current state of the FFI ($st'$.ffi). Function `do_app` fails if not enough or too many arguments to operator *op* are given in *vs*. Therefore the semantics raises a type error (`Rabort Rtype_error`) if `do_app` fails. If successful, function `do_app` returns a new state for the global references (*refs*), a new state of the FFI (*ffi*), and a value $v$. The overall result of the `evaluate` call is then the global state updated with *refs* and *ffi*, and value $v$.

### Relaxed Floating-Point Semantics

Both the relaxed floating-point semantics and Icing use value trees to represent floating-point values. However, Icing's nondeterministic semantics cannot be directly added to CakeML source, because `evaluate` is a deterministic function. RealCake instead encodes the nondeterminism as a deterministic *optimization oracle*. Specifically, RealCake's relaxed floating-point semantics extends the global state with a floating-point optimization oracle:

```
fpState = <| rewrites : optimization list; opts : num → rewriteApp list;
            canOpt : optChoice; choices : num|>
```

The oracle stores the currently allowed optimizations in the field `rewrites`. Component `opts` encodes the oracle decisions of when which optimization is applied. `opts 0` returns all optimizations that are applied next during evaluation of a floating-point expression. The optimization scope `canOpt` models the fine-grained control by recording the last optimization scope that has been seen while evaluating. The relaxed floating-point semantics optimizes only if `canOpt` is an `opt`:annotation. In `choices` we track the number of optimizations that have been applied. We will use this global counter for integrating the relaxed floating-point semantics with the proof-producing synthesis.

In principle, RealCake's relaxed floating-point semantics and the Icing semantics model the same set of optimization results, as for each nondeterministic Icing result there exists a deterministic oracle under which RealCake's semantics returns the same value, and vice versa. However, supporting floating-point optimizations in CakeML source is only possible with the optimization oracle. Adding the oracle to the global state of the semantics causes the least amount of friction with existing CakeML proofs, while also enabling the nondeterministic simulation proofs from Icing in CakeML via manipulation of the global optimization oracle.

To integrate the relaxed floating-point semantics of RealCake with `evaluate`, Figure 5c adds a separate case to `evaluate` for floating-point operators. As for standard operator evaluation in Figure 5a, when evaluating a floating-point operation, `evaluate` first evaluates the arguments, and runs `do_app`. When evaluating a floating-point operation, function `do_app` does not alter the global state ($st'$.refs), and it does not call into the foreign function interface ($st'$.ffi). It simply returns the value tree representing operator *op* applied to argument values in *vs*. If `do_app` successfully returns value tree $r$, `evaluate` attempts to optimize the value tree. To this end, function `optimizeIfOk` first checks whether the `canOpt` field of the optimization oracle is set to `opt`. If optimizations are allowed, the function performs the optimizations of the oracle in field `opts`. Then, the optimization oracle is advanced to the next decision, and the global optimization counter `choices` is incremented. Function `optimizeIfOk` returns both the global state updated with the new optimization oracle, and the optimized value tree. If no optimizations are allowed, the function leaves its inputs unchanged. Finally, if `op` is a Boolean comparison of floating-point value trees (`isFpBool op`), `evaluate` turns the resulting value tree into a Boolean constant as CakeML eagerly evaluates control-flow expressions.

For the loose connection between Icing and CakeML, it was sufficient for Icing to turn value trees into floating-point words once a control-flow decision was made. To keep the changes to the CakeML semantics local and manageable, RealCake's relaxed floating-point semantics eagerly evaluates value trees into words as soon as a Boolean comparison is applied to them, even if no control-flow decision is made afterwards.

Figure 5b adds the optimization annotations `opt:` and `noopt:` as a separate case to `evaluate`. `FpOptimize annot e` means that expression $e$ is evaluated under the optimization scope *annot*, which can either be `opt:` or `noopt:`. Evaluation of an optimization scope replaces the current semantic scope in `canOpt` with the new scope annotation (`updateOptFlag st annot`), before evaluating $e$. Next, the old annotation is recovered by `resetOptFlag` $st'$ $st$. Function `addAnnot` *annot vs* ensures that all value trees in `vs` are extended with a correct scoping annotation. This is required to ensure that the semantics respects the fine-grained control. If `evaluate` did not add the annotation to the value trees, the semantics could optimize expression `noopt:(x + 2.4)` by first evaluating `x + 2.4` and then optimizing it once the expression is used as part of a larger floating-point expression.

## 4.3 Integrating Relaxed Floating-Point Semantics into the Compiler Toolchain

If we want to fully integrate RealCake's relaxed floating-point semantics with CakeML, we have to also integrate it with the tools included in the CakeML compiler toolchain. In the toolchain, a binary implementation of the compiler is obtained by verified bootstrapping [29] of the in-logic compiler using proof-producing synthesis [2]. Furthermore, CakeML source code can be verified using CakeML's program verification tools that rely on characteristic formulae (CF) [22], allowing Hoare-logic like manual proofs (*e.g.*, for verification of non-terminating programs [46] or a proof checker for higher-order logic [1]). To prove whole-program specifications (Section 6), we integrate RealCake's relaxed floating-point semantics with the proof-producing synthesis and CF.

### CakeML Compiler Backend

A key insight for getting the deterministic compiler proofs to interact nicely with the optimization oracles used in RealCake's relaxed floating-point semantics was to implement the fast-math optimizer as a *source-level* optimization pass, separate from the CakeML compiler backend. With our extension from Subsection 4.1, the CakeML compiler backend compiles deterministic 64-bit floating-point kernels to machine code and we reuse this infrastructure by adding a third optimization scope, `strict`, to the relaxed floating-point semantics. Intuitively, we use the `strict` annotation to completely disallow floating-point optimizations in the compiler backend, allowing us to preserve determinism of the source semantics for the correctness proofs.

Any program that is run with the `strict` annotation will never apply optimizations and perform only IEEE-754 correct arithmetic operations. The difference between a `strict` and a `noopt` annotation is that `strict` is "sticky" in the sense that if a program ever enters `strict` mode, evaluation becomes deterministic and cannot escape from it through successive `opt` annotations, while `noopt` and `opt` can be mixed freely; e.g. a program may be under a `noopt` scope, while parts of it are marked with `opt` to selectively apply optimizations.

### Proof-Producing Synthesis and CF

The proof-producing synthesis and the CF are key components of the CakeML compiler, and required for bootstrapping the compiler. As both crucially depend on how the CakeML source semantics are defined, we have to make sure that the bootstrapping still works, even after adding the relaxed floating-point semantics. Specifically, the synthesis relies on expressions being pure, and thus not altering global state. The crux is that we need the optimization oracle to reside in global state for the backwards simulation proofs, and therefore must ensure

that no floating-point optimizations can be applied in code produced by synthesis. The `choices` component of the optimization oracle makes optimization attempts by the semantics observable in the global state. We prove a lemma that if the optimization counter `choices` is not incremented, evaluation cannot have attempted to optimize floating-point code under an `opt`: scope. To use this lemma, the synthesis configures the initial optimization oracle to be running under an `opt` scope, with an empty list of optimization choices. From this configuration we show that no floating-point optimizations are ever attempted by synthesized code, reestablishing the invariant of the expression being pure.

We use an optimization counter instead of a Boolean flag, as some of our simulation theorems must combine optimization oracles, while preserving optimization decisions (*e.g.*, when combining oracles for left and right-hand sides of binary operators). In such proofs, the optimization counter gives an exact bound on when the behavior of the oracle must change.

The exact same technique is applied to CF: we make sure that programs reasoned about with CF cannot apply optimizations based on the optimization oracle.

## 4.4 Extending CakeML with Real-Number Arithmetic

The third semantics added to CakeML in RealCake is a real-number semantics used for bounding roundoff errors of floating-point kernels. We extend the CakeML source semantics with support for real numbers and real-number operations by adding a new case to `evaluate`'s operator evaluation in Figure 5d. Here, we focus on the real-number semantics. In Section 6 we explain how RealCake translates floating-point programs into their real-number counterpart.

Evaluation of real-number operations follows the simple case from Subsection 4.2. The main difference is that we extend the optimization oracle in the global state with an additional flag `real_sem`. Function `realsAllowed st.fp_state` checks that the flag is set to true, otherwise evaluation is aborted. The flag disallows real-number operations where necessary, as the real-valued semantics is only used for verification purposes. Further, the compiler does not compile real-valued operations or constants. In the compiler proofs, we rule out real-number operations by assuming that the flag is switched off.

We have presented operator evaluation separately. In our implementation, when evaluating an `App` *op es* expression, the CakeML source semantics first does a case split on *op* and chooses whether to apply standard operator evaluation (Figure 5a), relaxed floating-point semantics (Figure 5c), or real-number semantics (Figure 5d).

When integrating the relaxed floating-point semantics with proof-producing synthesis of CakeML (Subsection 4.3), the global counter `choices` is used to make attempted floating-point optimizations observable. To preserve invariants of the proof-producing synthesis, the real-number semantics requires a similar treatment: The global counter `choices` is incremented if evaluation of a real-number operation is attempted but fails (`advanceOracle`).

# 5 RealCake's Floating-Point Optimizer

In this section, we implement a fast-math-style peephole optimizer for RealCake and prove it correct with respect to the relaxed floating-point semantics. At a high-level, we split optimization into two steps. In step one, function `planOpts` computes which optimizations should be applied to the kernel. We call the list of optimizations returned by `planOpts` the *optimization plan* and refer to this first step as *optimization planning*. In step two, function `applyOpts(plan,e)` applies the optimization plan `plan` to floating-point kernel `e`. Before returning, the `noOpts` function tags the result with a marker to dissallow further optimizations, which is required to recover the determinism needed by the CakeML compiler proofs. We call this second step *optimization execution*.

$$x \times 0 \to 0 \qquad x \times 2 \to x + x^* \qquad -(x \times y) \to x \times (-y)^*$$
$$x \times 1 \to x \qquad x \times 3 \to x + (x + x) \qquad x + (-y) \to x - y^*$$
$$x \times -1 \to -x \qquad x + 0 \to x \qquad x \times y + z \to \texttt{fma}(x, y, z)$$
$$x - x \to 0$$

**Figure 6** Optimizations currently used by the peephole optimization phase, IEEE-754 preserving optimizations are marked with a $*$.

## Optimization Planning

For a floating-point kernel `e`, function `planOpts(e)` returns a list of tuples `(path, opts)`, where the left-hand side `path` is an index into the kernel stating *where* the kernel should be optimized, and the right-hand side `opts` is a list of optimizations stating *how* the kernel should be optimized. The optimization planner `planOpts` is split into the following phases (applied in this order):

- `canonicalForm` puts all floating-point kernels into a canonical shape replacing $x - y$ with $x + ((-1) \times y)$, associating $+, \times$ to the right $((x + y) + z \to x + (y + z))$, and moving constants to right-hand sides with commutativity of $+$ and $\times$.
- `undistribute` replaces expressions like $(x \times y) + (x \times z)$ with $x \times (y + z)$, "undistributing" as much as possible to increase possibilities for FMA-introduction, and reduce the size of the floating-point kernel. The symmetric case of $(y \times x) + (z \times x)$ is ignored by the undistribute phase, as `canonicalForm` rotates all multiplications with commutativity.
- `peepholeOptimize` re-establishes canonical form and applies the optimizations from Figure 6.
- `balanceTrees` reorders sub-expressions in the floating-point kernel by replacing deeply-nested arithmetic expressions like $x_1 + (x_2 + (x_3 + x_4))$ by more shallow versions, such as $(x_1 + x_2) + (x_3 + x_4)$ and similarly for $\times$.[8]

Function `composePlans` concatenates the optimization plans produced by each phase.

## Optimization Execution

When executing the optimization plan, function `applyOpts` first runs function `optimizeWithPlan` on the plan and its input kernel, where `optimizeWithPlan` applies all elements of a given optimization plan one by one. Function `optimizeWithPlan` optimizes an expression only if it is wrapped under an `opt:` annotation. Further, either all or none of the optimizations in the plan are applied: if optimization fails, then the unoptimized input kernel is returned.

For each element of the plan `(path, opts)`, `optimizeWithPlan` traverses expression `e` following `path` until reaching a sub-expression `e'` and applies the optimizations `opts` at the end of the path. Having reached expression `e'` at the end of `path`, function `optimizeWithPlan` calls function `rewrite(e, opts)` that applies the optimizations `opts` to the CakeML expression `e'`.

As CakeML source supports stateful features like reference cells, and calls into a foreign-function-interface (FFI), function `rewrite(e, opts)` checks that CakeML expression `e` is a pure (floating-point) expression. This check, which is implemented as a function `isPureExp(e)`, effectively rules out optimization of expressions that use any of CakeML's stateful features.

[8] We added `balanceTrees` as an optimization pass to simplify register allocations.

The result of running `optimizeWithPlan` is given to function `noOpts`, that performs a bottom-up traversal of expression `e`, replacing any `opt:` annotation with a `noopt:` annotation, disallowing further optimizations and, as a result, making the program's semantics deterministic.

## 5.1 Correctness of the Fast-Math Optimizer

Our optimizer is split into two separate phases, optimization planning, and optimization execution. A key benefit of this split is that we can prove correctness of optimization execution without caring about the exact optimizations contained in the plan. Rather, we verify `applyOpts` for any potential plan generated by our optimization planner.

At a high-level, we show that the optimizations done by `applyOpts` are correct with respect to the relaxed floating-point semantics, and no further optimizations can be applied afterwards. Accordingly, we split correctness of `applyOpts` into two proofs. First, we prove that running the result of `noOpts`(e) gives the same result as running `e` with an oracle that performs no optimizations. The correctness proof for `noOpts`(e) is a simple backwards simulation and thus we do not show it here. Second, we prove that there is a backwards simulation between the result of `optimizeWithPlan(e, plan)` and `e`, where `plan` has been generated by our planner.

▶ **Theorem 2** (`optimizeWithPlan` – correctness).

$$
\begin{aligned}
&\texttt{evaluate}\ st_1\ env\ (\texttt{optimizeWithPlan}\ (\texttt{planOpts}\ e)\ exps)\ =\ (st_2, \texttt{Rval}\ r)\ \wedge\\
&\texttt{allVarsBoundToFPVal}\ \ exps\ env \wedge\ \texttt{flagAndScopeAgree}\ cfg\ st_1.\texttt{fp\_state}\ \wedge\\
&\texttt{notInStrictMode}\ st_1.\texttt{fp\_state}\ \wedge\ \texttt{noRealsAllowed}\ st_1.\texttt{fp\_state}\ \Rightarrow\\
&\exists fpOpt\ choices\ fpOptR\ choicesR.\\
&\ \texttt{evaluate}\ (\texttt{appendOptsAndOracle}\ st_1\ (\texttt{getRws}\ (\texttt{planOpts}\ e))\ fpOpt\ choices)\ env\ exps\ =\\
&\ \ (\texttt{appendOptsAndOracle}\ st_2\ (\texttt{getRws}\ (\texttt{planOpts}\ e))\ fpOptR\ choicesR, \texttt{Rval}\ r)
\end{aligned}
$$

Theorem 2 proves: for the result obtained from evaluating the syntactically optimized kernel, there exists an optimization oracle such that `evaluate` returns the same result when semantically optimizing with the optimizations from the computed plan. The CakeML source semantics is untyped, and thus we assume that all variables are bound to floating-point constants in `exps` (`allVarsBoundToFPVal`). Instead of showing correctness of `optimizeWithPlan` for the overall plan, we reduce the global correctness proof to a series of correctness proofs about the separate phases, and combine them into the overall backwards simulation.

### Extending the Optimizer

Extending the RealCake optimizer requires extending both the implementation of the optimizer, and its correctness proof. To add a new peephole optimization, a user adds the optimization to the list of optimization of `peepholeOptimize`, and extends the correctness theorem for `peepholeOptimize`. All other theorems need not be changed. We provide a set of lemmas that can be used to reduce the global correctness proof of `peepholeOptimize` to a simple local backwards simulation for the newly-added optimization in terms of the `rewrite` function only. Adding a new phase to `planOpts` is more involved as it requires showing a global correctness theorem for the newly added phase, as well as extending the theorem that splits up correctness of `planOpts` into correctness of its components. The complexity of the first proof depends on the complexity of the phase, whereas splitting up the correctness proof for `planOpts` is a straightforward proof showing that optimizations of the newly added phase are contained in the optimizations applied by `planOpts`.

# 6 Proving Error Refinement with RealCake

CakeML with relaxed floating-point semantics optimizes floating-point kernels and automatically proves a relation between the unoptimized and the optimized kernel. However, to meaningfully support floating-point arithmetic in a verified compiler, the compiler must relate the unoptimized real-valued program and the optimized floating-point program.

Classic compiler optimizations like constant propagation and dead-code elimination have a clear definition of when they can be applied and one can prove that the optimizations do not change the program result. Floating-point fast-math optimizations do not follow this intuition in general. As an example, we can introduce an FMA instruction in the simple expression `x * 2.9 + 0.05` with relaxed floating-point semantics: `fma(x, 2.9, 0.05)`. The FMA makes the expression generally faster and locally more accurate, as the result is only rounded once. Correctness of the fast-math optimizer proves a backwards simulation between the expressions, however, the theorem does not capture the change in roundoff errors.

We propose the notion of *error refinement*: the compiler may optimize a floating-point kernel aggressively as long as the results remain within a (given) bound relative to *real-number* semantics. This notion captures the implicit assumption or expectation by the programmer. We make this notion of error refinement explicit by implementing a fully automatic pipeline that computes an upper bound on the roundoff error of a floating-point kernel in CakeML and compares it to a user-specified accuracy bound. For this we use the roundoff error analysis tool FloVer [6], implemented in HOL4. We prove the roundoff error bound correct with respect to a run of the original input kernel under an idealized real-number semantics.

## 6.1 Translating RealCake Kernels into FloVer Input

To infer roundoff errors for a RealCake kernel with FloVer, we define a straightforward encoding function `toFloVer`(e), translating floating-point kernels with variables, constants, unary and binary floating-point operations, FMAs, and let bindings into FloVer syntax. Correctness of the translation functions proves once and for all a simulation relating deterministic RealCake floating-point semantics with FloVer's idealized finite-precision semantics. To prove the simulation, our translation function ensures that the kernel is wrapped under a `noopt:` annotation. As roundoff error analysis tools depend on ranges for the input variables our pipeline also requires a real-number function specifying these input constraints.

RealCake implements a function `isOkError(e, P, err)` that returns true if `err` is a sound upper bound on the worst-case roundoff error for RealCake expression `e` and input constraints `P`. First, the RealCake kernel `e` is translated into FloVer syntax with `toFloVer`(e). Function `isOkError` then runs FloVer's unverified inference algorithm to generate a (untrusted) roundoff error analysis certificate for the FloVer encoding of `e` and input constraints `P`. FloVer's certificate checker automatically checks the certificate, and if the check suceeds, the error bound encoded in the certificate is correct. Finally, `isOkError` checks that the global upper bound encoded in the certificate is smaller or equal to the user-specified error constraint `err`.

## 6.2 Proving Roundoff Error Bounds for RealCake Kernels

To prove error refinement for an optimized kernel, we connect the soundness theorem of FloVer to RealCake's relaxed floating-point semantics. Together with the idealized real-valued semantics we show once and for all the HOL4 theorem:

▶ **Theorem 3** (CakeML-FloVer roundoff errors)**.**

$$
\begin{array}{l}
\forall f\ P\ err\ theVars\ vs\ body\ env.\\
\quad \texttt{isOkError\_succeeds}\ (f, P, err, theVars, body)\ \wedge\ \texttt{isPrecondFine}\ theVars\ vs\ P\ \Rightarrow\\
\quad \exists r\ fp.\\
\qquad \texttt{realEvals\_to}\ (\texttt{realify}\ body)\ (\texttt{envWithRealVars}\ env\ theVars\ vs)\ r\ \wedge\\
\qquad \texttt{floatEvals\_to}\ body\ (\texttt{envWithFloatVars}\ env\ theVars\ vs)\ fp\ \wedge\\
\qquad \texttt{abs}\ (\texttt{valueTree2real}\ fp\ -\ r)\ \leq\ err
\end{array}
$$

On a high-level, Theorem 3 states that if function `isOkError` succeeds, the analyzed function can be run both under floating-point and real-number semantics, and `err` is an upper bound on the roundoff error. The assumptions are: `isOkError` succeeds, and `body` is the function body of RealCake floating-point kernel `f`, with the parameters `theVars` (`isOkError_succeeds`); and the values *vs* bound to the parameters *theVars* are within the input constraints $P$ (`isPrecondFine` *theVars vs* $P$). `realify` replaces floating-point operations by their real-number counterparts. The theorem shows that there exists a real number $r$ and a floating-point word *fp* such that evaluation of the function under an idealized real-number semantics returns $r$ (`realEvals_to`), evaluation under floating-point semantics returns *fp* (`floatEvals_to`), and *err* is an upper bound to the roundoff error of function $f$ (`abs`(`valueTree2real` $fp - r) \leq err$).

Error refinement relates the user-given error bound back to a real-number semantics of the initial, unoptimized kernel, but RealCake runs function `isOkError` on the optimized kernel. In addition to Theorem 3 we also need to prove that the applied optimizations are real-valued identities. Exactly like we prove correctness of `optimizeWithPlan` in Subsection 5.1, we have proven once and for all a simulation between the real-number semantics of the optimized kernel and its unoptimized version. Combining this theorem with Theorem 3, we automatically prove error refinement for floating-point kernels.

## 7 Evaluation: Performance and Accuracy Proofs

The RealCake development spans roughly 35k lines of proof-code, composed of the IEEE floating-point implementation and proofs, including the ARMv7 backend (∼1.5k LOC), the relaxed floating-point semantics and the real-number semantics (∼7k LOC, including proofs), the implementation and correctness proofs for the optimizer (∼20k LOC), and the benchmarks from the evaluation (∼7k LOC).

We evaluate RealCake on 51 benchmarks taken from the standard floating-point benchmark set FPBench [14]. Our evaluation includes all FPBench benchmarks that use floating-point operations that are supported by RealCake and we exclude only those that cannot be expressed in RealCake (for instance we exclude benchmarks with elementary function calls; *i.e.* functions like sin, cos, ...). We use the preconditions that are already specified in FPBench, but modify them slightly for the `jetEngine` and `n_body` kernels such that FloVer can prove a roundoff error bound and does not report a possible division by zero. Our evaluation shows how RealCake establishes end-to-end correctness proofs, and compares the runtime of the optimized and unoptimized kernels.

**Table 1** Roundoff errors for optimized and unoptimized FPBench benchmarks; benchmarks where the roundoff error improves are highlighted in bold font and benchmarks where no end-to-end specification is proven are underlined.

| Name | Orig | fast-math | Impr. |
|---|---|---|---|
| bspline3 | 1.295e-16 | 1.295e-16 | 0% |
| carbonGas | 5.688e-08 | 5.688e-08 | 0% |
| **cartToPol** | 2.815e-09 | 2.463e-09 | 13% |
| **delta4** | 4.048e-12 | 2.028e-13 | 75% |
| delta | 1.970e-13 | 2.940e-12 | -198% |
| **doppler1** | 6.534e-13 | 6.412e-13 | 2% |
| **doppler2** | 6.534e-13 | 1.639e-12 | 50% |
| **doppler3** | 1.675e-12 | 2.680e-13 | 20% |
| **himmilbeau** | 3.417e-12 | 3.003e-12 | 12% |
| **hypot** | 2.815e-09 | 2.463e-09 | 13% |
| **hypot32** | 2.815e-09 | 2.463e-09 | 13% |
| i4modified | 4.002e-13 | 4.002e-13 | 0% |
| intro_ex | 2.220e-10 | 2.220e-10 | 0% |
| **jetEngineModi** | 5.209e-08 | 3.898e-08 | 25% |
| kepler0 | 1.761e-13 | 1.801e-13 | -2% |
| kepler1 | 8.397e-13 | 8.467e-13 | -1% |
| **kepler2** | 4.069e-12 | 3.973e-12 | 2% |
| **matDet2** | 5.107e-12 | 4.663e-12 | 9% |
| **matDet** | 5.107e-12 | 4.663e-12 | 9% |
| n_bodyXmod | ERR | ERR | ERR |
| n_bodyZmod | ERR | ERR | ERR |
| nonlin1 | 2.220e-10 | 2.220e-10 | 0% |
| nonlin2 | 2.657e-09 | 2.657e-09 | 0% |
| pid | 7.621e-15 | 7.727e-15 | -1% |
| **predatorPrey** | 3.395e-16 | 3.366e-16 | 1% |
| **rigidBody1** | 6.565e-11 | 5.329e-13 | 80% |

| Name | Orig | fast-math | Impr. |
|---|---|---|---|
| rigidBody2 | 5.579e-13 | 6.410e-11 | -360% |
| **rump_C** | 4.079e+22 | 3.859e+22 | 5% |
| **rump_rev** | 3.859e+22 | 3.679e+22 | 5% |
| **rump_pow** | 4.079e+22 | 3.859e+22 | 5% |
| runge_kutta_4 | 2.220e-14 | 2.220e-14 | 0% |
| sec4_example | 2.657e-09 | 2.657e-09 | 0% |
| **sine_newton** | 7.495e-15 | 6.275e-15 | 16% |
| sineOrder3 | 1.765e-15 | 1.765e-15 | 0% |
| **sine** | 1.538e-15 | 1.373e-15 | 11% |
| **sqroot** | 1.115e-15 | 1.059e-15 | 5% |
| sqrt_add | 1.322e-12 | 1.322e-12 | 0% |
| sum | 5.995e-15 | 5.995e-15 | 0% |
| t01_s3 | 5.995e-15 | 5.995e-15 | 0% |
| **t02_s8** | 9.548e-15 | 8.438e-15 | 12% |
| t03_nl2 | 4.885e-14 | 4.885e-14 | 0% |
| t04_dqmom9 | 1.999 | 1.999 | 0% |
| t05_nl1_r4 | 4.441e-06 | 4.441e-06 | 0% |
| t05_nl1_t2 | 2.776e-16 | 2.776e-16 | 0% |
| **t06_sums4_sum1** | 1.443e-15 | 1.332e-15 | 8% |
| t06_sums4_sum2 | 1.332e-15 | 1.332e-15 | 0% |
| **turbine1** | 1.588e-13 | 1.541e-13 | 3% |
| turbine2 | 2.213e-13 | 2.213e-13 | 0% |
| **turbine3** | 1.108e-13 | 1.061e-13 | 4% |
| verhulst | 8.343e-16 | 8.343e-16 | 0% |
| x_by_xy | 2.220e-15 | 2.220e-15 | 0% |

## 7.1 Automated End-To-End Proofs

We have translated all 51 FPBench benchmarks into HOL4 script files that are read by RealCake. Each script file defines the original, unoptimized, floating-point kernel, a precondition for the kernel, and a user-provided error bound. For simplicity, our evaluation uses $2^{-5}$ as the user-provided error bound for all of the benchmarks, though those would be given by the compiler user in a real-world setting.[9]

Our HOL4 automation at the end of each script file fully automatically optimizes the kernel, instantiates Theorem 2 for the generated plan, infers a roundoff error bound and compares it to the user-provided error bound. Finally, a whole-program specification relating the behavior of the machine code for the optimized program to the real-number semantics of the unoptimized program is proven automatically by combining the individual proofs.

RealCake proves the end-to-end correctness theorem (Theorem 1) for 45 benchmarks. That is, for these benchmarks it is able to show that the roundoff error of the optimized program is below the specified default error bound of $2^{-5}$. For the three `rump` benchmarks and the `test04_dqmom9` benchmark, the computed errors are larger than the user-provided error bound (already for the original unoptimized program), and for the benchmarks `n_bodyXmod` and `n_bodyZmod` FloVer is not able to infer a roundoff error bound as its HOL4 computation becomes stuck, likely due to limitations in the HOL4 real number computations.

[9] If the error bound is choosen too tightly the optimizer may reject every optimization candidate, while a too coarse bound could allow for too aggressive optimizations.

**Table 2** Running times for optimized and unoptimized FPBench benchmarks on the Raspberry Pi v3; benchmarks where performance improves *with fast-math optimizations* are highlighted in bold.

| Name | Orig | Csts | Csts + fast-math |
|---|---|---|---|
| bspline3* | 18.14 | 1.75 (91%) | 1.75 (91% / 0%) |
| carbonGas * | 103.40 | 3.85 (97%) | 3.85 (97% / 0%) |
| **cartToPol** | 2.05 | 2.04 (1%) | 1.86 (10% / 9%) |
| **delta4** | 6.34 | 6.33 (1%) | 6.17 (3% / 3%) |
| **delta** | 13.49 | 13.47 (1%) | 11.44 (16% / 16%) |
| **doppler1** | 36.02 | 3.25 (91%) | 3.06 (92% / 6%) |
| **doppler2** | 36.00 | 3.25 (91%) | 3.06 (92% / 6%) |
| **doppler3** | 35.98 | 3.25 (91%) | 3.07 (92% / 6%) |
| **himmilbeau** | 36.13 | 3.36 (91%) | 3.05 (92% / 10%) |
| **hypot32** | 2.04 | 2.04 (1%) | 1.86 (10% / 9%) |
| **hypot** | 2.05 | 2.05 (1%) | 1.86 (10% / 10%) |
| i4modified* | 1.77 | 1.78 (0%) | 1.78 (0% / 0%) |
| intro_ex* | 17.73 | 1.32 (93%) | 1.32 (93% / 0%) |
| **jetEngineMod** | 195.99 | 11.89 (94%) | 11.12 (95% / 7%) |
| kepler0 | 5.32 | 5.31 (1%) | 5.30 (1% / 1%) |
| kepler1 | 8.19 | 8.20 (0%) | 8.16 (1% / 1%) |
| **kepler2** | 12.43 | 12.41 (1%) | 12.22 (2% / 2%) |
| **matDet2** | 6.38 | 6.37 (1%) | 5.67 (12% / 12%) |
| **matDet** | 6.37 | 6.38 (0%) | 5.65 (12% / 12%) |
| **n_bodyXmod** | 38.46 | 5.20 (87%) | 5.06 (87% / 3%) |
| n_bodyZmod | 38.40 | 5.27 (87%) | 5.15 (87% / 0%) |
| nonlin1* | 17.72 | 1.31 (93%) | 1.31 (93% / 0%) |
| nonlin2* | 35.06 | 2.41 (94%) | 2.41 (94% / 0%) |
| pid* | 104.11 | 4.72 (96%) | 4.72 (96% / 0%) |
| predatorPrey | 52.25 | 2.81 (95%) | 3.08 (95% / -9%) |
| rigidBody1* | 19.11 | 2.78 (86%) | 2.78 (86% / 0%) |

| Name | Orig | Csts | Csts + fast-math |
|---|---|---|---|
| **rigidBody2** | 54.92 | 5.10 (91%) | 4.54 (92% / 11%) |
| **rump_C** | 107.48 | 6.82 (94%) | 6.26 (95% / 9%) |
| **rump_rev** | 107.96 | 6.80 (94%) | 6.27 (95% / 8%) |
| **rump_pow** | 112.54 | 12.34 (90%) | 11.57 (90% / 7%) |
| runge_kutta_4* | 93.46 | 9.53 (90%) | 9.53 (90% / 0%) |
| sec4_example* | 34.99 | 2.40 (94%) | 2.40 (94% / 0%) |
| sineOrder3* | 34.86 | 2.08 (95%) | 2.08 (95% / 0%) |
| sine_newton* | 126.34 | 10.73 (92%) | 10.73 (92% / 0%) |
| sine* | 55.36 | 6.03 (90%) | 6.03 (90% / 0%) |
| **sqroot** | 87.06 | 4.85 (95%) | 4.65 (95% / 5%) |
| sqrt_add* | 35.21 | 2.59 (93%) | 2.59 (93% / 0%) |
| sum* | 3.07 | 3.07 (1%) | 3.07 (1% / 0%) |
| t01_s3* | 3.07 | 3.08 (0%) | 3.08 (0% / 0%) |
| t02_s8* | 3.04 | 3.05 (0%) | 3.05 (0% / 0%) |
| t03_nl2* | 1.78 | 1.78 (1%) | 1.78 (1% / 0%) |
| **t04_dqmom9** | 163.82 | 11.76 (93%) | 10.20 (94% / 14%) |
| t05_nl1_r4* | 34.67 | 2.06 (95%) | 2.06 (95% / 0%) |
| t05_nl1_test2* | 34.00 | 1.54 (96%) | 1.54 (96% / 0%) |
| t06_sums4_sum1* | 1.70 | 1.70 (0%) | 1.70 (0% / 0%) |
| t06_sums4_sum2* | 1.68 | 1.67 (1%) | 1.67 (1% / 0%) |
| turbine1* | 121.02 | 5.29 (96%) | 5.29 (96% / 0%) |
| turbine2* | 69.90 | 3.94 (95%) | 3.94 (95% / 0%) |
| turbine3* | 121.29 | 5.28 (96%) | 5.28 (96% / 0%) |
| verhulst* | 51.28 | 2.27 (96%) | 2.27 (96% / 0%) |
| x_by_xy* | 1.51 | 1.51 (1%) | 1.51 (1% / 0%) |

We show the errors for the optimized and unoptimized kernels in Table 1. "Orig." is the roundoff error for the unoptimized kernel, and "fast-math" is the roundoff error for the optimized kernel, and column "Impr." shows the percentage by which the error improved with our fast-math optimizations, *i.e.* if the number is less than 0% the error has increased, and decreased if it is greater than 0%. We highlight benchmarks where the roundoff error has been decreased by the RealCake optimizer in bold font. While improving the roundoff error is not the goal of our optimizations, FMA instructions are said to be locally more accurate, and reordering of operations influences roundoff errors too. Hence we evaluate the effect on roundoff errors of our optimization strategy. The benchmarks `delta4`, `delta`, `rigidBody1`, and `rigidBody2` have the largest difference in roundoff errors. By inspecting the generated code we found that in these cases, RealCake has significantly alterted the structure of the kernel. The roundoff error computed for a single kernel is highly influenced by the order of operations, thus we suspect that this large difference is mainly due to operator odering. Overall, we notice that if RealCake can infer a roundoff error, the error of the optimized kernel is usually within the same order of magnitude as the unoptimized version, but in many cases it is actually more accurate.

## 7.2 Performance Improvements

We compared the performance of unoptimized and RealCake's optimized floating-point kernels. In a first run, we measured wide differences in speedups and slowdowns. By manually inspecting the code, we noticed a missing optimization in CakeML: 64-bit word constants should be pre-allocated (or lifted) to increase performance. Lifting constants is a worthwhile optimization in general, and particularly effective for floating-point programs, as

it is does not change the program's IEEE-754-semantics and floating-point programs usually contain many constants. Thus, we implemented an independent, semantics preserving, global optimization that preallocates 64-bit words as global variables. Our performance evaluation compares three versions of FPBench kernels: the unoptimized version as a baseline, the kernel with preallocated constants, and the kernel after first applying fast-math optimizations and then preallocating constants.

To measure performance, CakeML generates ARMv7 machine code where each numerical kernel is run 10 million times in a loop. Each version of the benchmark, with the core loop running the kernel 10 million times, is run three times on five different sets of inputs, for a total of fifteen runs per benchmark.

We run the ARMv7 code on a Raspberry Pi v3 and summarize the results in Table 2. Column "Orig." shows the running time of the (10 million iterations of the) unoptimized program in seconds. Column "Csts." shows the running time of the program with preallocated constants in seconds plus the relative speedup in percent. And column "Csts. + fast-math" shows the running time of the program when first running RealCake's optimizer and then preallocating constants in seconds plus first the relative speedup in percent with respect to the unoptimized program, and second the relative speedup with respect to the version with preallocated constants. We mark benchmarks with a performance improvement of more than 1% of the fast-math optimizations with respect to preallocating constants in bold (we identified a difference within ±1% to be noise).

Initially, some benchmarks experienced slowdowns of up to 20%. Via manual inspection, we noticed that the fast-math optimizer created too many instructions. As a simple heuristic to prevent this problem, RealCake sums the arities of the floating-point operators in the program versions, and returns the unoptimized version if the heuristic value of the fast-math optimized program is greater than or equal to the unoptimized program. Even if the heuristic rejects an optimization, RealCake computes roundoff errors for both program versions and proves an end-to-end specification theorem about the optimized program. In total, the heuristic rejects optimizations for 27 benchmarks, and we mark them with a $*$ in Table 2.

Overall the evaluation shows that preallocating constants is a valuable optimization for CakeML on its own. On top of this, our fast-math optimizer is able to improve the performance for 20 benchmarks and for 7 of those significantly ($> 10\%$). This is remarkable, since the FPBench benchmarks are carefully hand-written and do not target optimizations specifically and are not representative of, e.g., automatically generated code from tools such as Matlab that would be used in the development of embedded system kernels.

For one benchmark we notice a slowdown of 9% even with our heuristic, and the program versions differ only by a single FMA instruction. We suspect that this slowdown is due to bad pipelining on the Raspberry Pi.

RealCake's constant preallocation achieves a geometric mean speeup of 83%, and the geometric mean of the speedup for RealCake's optimizer compared with the program with preallocated constants is 3%. The maximum speedup achieved with preallocating constants only is 97%, and we notice no slowdowns. When applying fast-math optimization, the greatest slowdown is -9%, and the maximum speedup is 16%.

In general, benchmarks with higher speed-ups from our optimization strategy usually provide many opportunities to both introduce `fma` instructions, and remove constants. We think that the foundational work in RealCake facilitates exploration of other optimization strategies in the future.

# 8 Related Work

## Verified Compilation of Floating-Point Programs

Besides CakeML, CompCert [31] is the other major available verified compiler, compiling imperative C programs. CompCert supports floating-point programs [10] following the strict IEEE-754 semantics. This semantics allows it to perform a few small optimizations that are IEEE-754 compliant such as constant propagation and replacing a multiplication by two by an addition ($x \times 2 \rightarrow x + x$).

RealCake supports additional optimizations based on real-valued identities that are not IEEE-754 compliant. While our implementation is done in the context of CakeML and verified in HOL4, the principles of RealCake are independent of the particular programming language that is being compiled and should thus be portable to CompCert as well.

The Alive framework [34] provides a way to specify and prove correct peephole optimizations for C++ code that can be applied in an LLVM pass. Alive verifies optimizations using SMT solvers and has been extended to bit-precise floating-point optimizations and optimizations involving special values, satisfying the IEEE-754 standard [39, 42]. These optimizations are complementary to RealCake's optimizations. Formal verification of Alive's peephole optimizations is addressed separately by the AliveInLean project [30]. The VELLVM project [59] provides a rigorous semantics for LLVM IR semantics to reason about optimizations and implements IEEE-754-preserving floating-point arithmetic.

## Verification of Floating-point Programs

Besides FloVer, there are several other tools that provide formally verified roundoff error bounds for floating-point arithmetic expressions:
FPTaylor [54], real2Float [36], Precisa [41], Gappa [17], and each of these can in principle replace FloVer in RealCake. We chose FloVer for convenience as it is implemented in HOL4.

Verification of floating-point programs that go beyond numerical kernels is still relatively limited. The above-mentioned automated tools, for instance, do not consider function calls, and techniques for loops are very restricted [41, 15], and thus require the user to provide range annotations for each function call, as well as loop invariants in general. Entire programs have been manually formally verified w.r.t. a real-valued specification, but with considerable human effort [48, 9], which is not suitable for a compilation setting.

If we only require verification of runtime exceptions, resp. absence of special values, then abstract interpretation-based techniques do scale to larger programs [27] and some provide formal verification [28].

## Optimization of Floating-Point Programs

Floating-point optimizations have also been considered outside of the traditional compiler context, most of them focused on performance optimization.

Precimonious [50] performs mixed-precision tuning, by determining which operations can be implemented in a lower or higher precision, while satifying a user-provided error bound. While Precimonious can handle short programs with loops, it cannot guarantee the error bound as it uses a dynamic error analysis. Both FPTuner [11] and Daisy [16] perform mixed-precision tuning while providing accuracy guarantees using a static analysis, but can only handle relatively short straight-line expressions. Mixed-precision tuning requires a global error analysis and is thus not suitable to be performed inside a fundamentally modular

compiler. However, the precision-tuned program could be further optimized by a (verified) compiler. While RealCake currently only supports double precision floating-point arithmetic, an extension with (uniform) single precision requires merely some engineering work.

Several tools improve the performance or accuracy of floating-point programs by rewriting with real-valued identities. Spiral [47] rewrites linear algebra kernels to improve their performance on a particular hardware platform. Spiral does not take into account roundoff errors; its rewrites are not IEEE-754-preserving, but it does not quantify the errors. The HELIX project [58] uses Spiral as an external oracle for building a verified optimization pipeline for dataflow optimizations. Optimizations in HELIX are done with respect to real-number semantics which is orthogonal to RealCake's floating-point peephole optimizer.

Herbie [44] aims to improve the accuracy, instead of performance, of floating-point arithmetic expressions but estimates roundoff errors unsoundly using a dynamic analysis. The Salsa [13] tool applies a set of transformation rules to improve performance while soundly tracking roundoff errors. Finally, Daisy [16] first applies rewriting similar to Herbie in order to improve performance gains due to mixed-precision tuning. Still further away is the tool STOKE [52], which generates small floating-point kernels by superoptimization, but which does not even guarantee real-valued equivalence. We consider these optimizations to be orthogonal to the fast-math optimizations that we consider in RealCake. We note that the scoping mechanism allows RealCake to easily integrate parts of the code that have been heavily optimized, and that thus should not be modified further by the compiler.

## 9 Conclusion

We have presented RealCake, an extension of the CakeML compiler with fast-math-style floating-point optimizations. Using an oracle-based relaxed floating-point semantics we have integrated nondeterminstic semantics for fast-math-style optimizations into the verified CakeML compiler. Via a connection to an external accuracy analysis, RealCake establishes accuracy guarantees for the optimized program, relating it back to the real-numbered execution of the unoptimized program. In summary, RealCake is the first verified compiler that establishes end-to-end floating-point accuracy guarantees to enable fast-math-style optimization and prove end-to-end compilation theorems. Our evaluation has shown how RealCake automatically verifies whole programs, proving properties about their I/O behavior and accuracy. Further, both our fast-math optimizer and our global constant lifting achieve significant performance improvements. RealCake's error refinement establishes the core infrastructure necessary to verify fast-math-style peephole optimizations, enabling the implementation of additional optimizations such as vectorization in the future.

## References


1 Oskar Abrahamsson. A Verified Proof Checker for Higher-Order Logic. *Journal of Logical and Algebraic Methods in Programming*, 112, 2020. doi:10.1016/j.jlamp.2020.100530.

2 Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. Proof-Producing Synthesis of CakeML from Monadic HOL Functions. *J. Autom. Reason.*, 64(7), 2020. doi:10.1007/s10817-020-09559-8.

3 A. Anta and P. Tabuada. To Sample or not to Sample: Self-Triggered Control for Nonlinear Systems. *IEEE Transactions on Automatic Control*, 55(9):2030–2042, 2010. doi:10.1109/TAC.2010.2042980.

4 Apache Software Foundation. The LLVM Compiler Infrastructure, 2020. URL: https://www.llvm.org/.

5 Heiko Becker, Eva Darulova, Magnus O Myreen, and Zachary Tatlock. Icing: Supporting Fast-Math Style Optimizations in a Verified Compiler. In *Computer Aided Verification (CAV)*, 2019. doi:10.1007/978-3-030-25543-5_10.

6 Heiko Becker, Nikita Zyuzin, Raphaël Monat, Eva Darulova, Magnus O Myreen, and Anthony Fox. A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4. In *Formal Methods in Computer Aided Design (FMCAD)*, 2018. doi:10.23919/FMCAD.2018.8603019.

7 Michael Berg. LLVM Numerics Blog, 2019. URL: http://blog.llvm.org/2019/03/llvm-numerics-blog.html.

8 Hans-J. Boehm. Towards an API for the Real Numbers. In *Programming Language Design and Implementation (PLDI)*, 2020. doi:10.1145/3385412.3386037.

9 Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave Equation Numerical Resolution: A Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning*, 50(4), 2013. doi:10.1007/s10817-012-9255-4.

10 Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified Compilation of Floating-Point Computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015. doi:10.1007/s10817-014-9317-x.

11 Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Rigorous Floating-Point Mixed-Precision Tuning. In *Principles of Programming Languages (POPL)*, 2017. doi:10.1145/3009837.3009846.

12 G. A. Constantinides, P. Y. K. Cheung, and W. Luk. Wordlength Optimization for Linear Digital Signal Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(10), 2003. doi:10.1109/TCAD.2003.818119.

13 Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. Improving the Numerical Accuracy of Programs by Automatic Transformation. *International Journal on Software Tools for Technology Transfer*, 19(4), 2017. doi:10.1007/s10009-016-0435-0.

14 Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In *Numerical Software Verification (NSV)*, 2016. doi:10.1007/978-3-319-54292-8_6.

15 Eva Darulova and Viktor Kuncak. Towards a Compiler for Reals. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(2), 2017. doi:10.1145/3014426.

16 Eva Darulova, Saksham Sharma, and Einar Horn. Sound Mixed-Precision Optimization with Rewriting. In *International Conference on Cyber-Physical Systems (ICCPS)*, 2018. doi:10.1109/ICCPS.2018.00028.

17 Florent De Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Assisted Verification of Elementary Functions Using Gappa. In *ACM Symposium on Applied Computing (SAC)*, 2006. doi:10.1145/1141277.1141584.

18 Anthony Fox. Improved Tool Support for Machine-Code Decompilation in HOL4. In *International Conference on Interactive Theorem Proving*. Springer, 2015. doi:10.1007/978-3-319-22102-1_12.

19 Free Software Foundation. The GNU Compiler Collection, 2020. URL: https://gcc.gnu.org/.

20 GCC Developers. GCC Wiki: Floating-point Math, 2020. URL: https://gcc.gnu.org/wiki/FloatingPointMath.

21 Alejandro Gomez-Londono, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. Do You Have Space for Dessert? A Verified Space Cost Semantics for CakeML Programs. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 4, 2020. doi:10.1145/3428272.

22 Armaël Guéneau, Magnus O Myreen, Ramana Kumar, and Michael Norrish. Verified Characteristic Formulae for CakeML. In *European Symposium on Programming (ESOP)*, 2017. doi:10.1007/978-3-662-54434-1_22.

**23** Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning (ICML)*, 2015.

**24** John Harrison. Floating Point Verification in HOL. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, 1995. doi:10.1007/3-540-60275-5_65.

**25** Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2nd edition, 2002. doi:10.1137/1.9780898718027.

**26** IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 2019. doi:10.1109/IEEESTD.2019.8766229.

**27** Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification (CAV)*, 2009. doi:10.1007/978-3-642-02658-4_52.

**28** Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A Formally-Verified C Static Analyzer. In *Principles of Programming Languages (POPL)*, 2015. doi:10.1145/2676726.2676966.

**29** Ramana Kumar. *Self-Compilation and Self-Verification*. PhD thesis, University of Cambridge, 2015.

**30** Juneyoung Lee, Chung-Kil Hur, and Nuno P Lopes. AliveInLean: a Verified LLVM Peephole Optimization Verifier. In *International Conference on Computer Aided Verification*. Springer, 2019. doi:10.1007/978-3-030-25543-5_25.

**31** Xavier Leroy. Formal Certification of a Compiler Back-end, or: Programming a Compiler with a Proof Assistant. In *Principles of Programming Languages (POPL)*, 2006. doi:10.1145/1111037.1111042.

**32** Xavier Leroy. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*, 43(4), 2009. doi:10.1007/s10817-009-9155-4.

**33** LLVM Developers. LLVM Language Reference: Fast-Math Flags, 2020. URL: https://llvm.org/docs/LangRef.html#fast-math-flags.

**34** Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably Correct Peephole Optimizations with Alive. In *Programming Language Design and Implementation (PLDI)*, 2015. doi:10.1145/2737924.2737965.

**35** Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. Verified Compilation on a Verified Processor. In *Programming Language Design and Implementation (PLDI)*, 2019. doi:10.1145/3314221.3314622.

**36** Victor Magron, George Constantinides, and Alastair Donaldson. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Transactions on Mathematical Software*, 43(4), 2017. doi:10.1145/3015465.

**37** Rupak Majumdar, Indranil Saha, and Majid Zamani. Synthesis of Minimal-Error Control Software. In *International Conference on Embedded Software (EMSOFT)*, 2012. doi:10.1145/2380356.2380380.

**38** Adolfo Anta Martinez, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. Automatic Verification of Control System Implementations. In *International Conference on Embedded software (EMSOFT)*, 2010. doi:10.1145/1879021.1879024.

**39** David Menendez, Santosh Nagarakatte, and Aarti Gupta. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *Static Analysis Symposium (SAS)*, 2016. doi:10.1007/978-3-662-53413-7_16.

**40** Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009. doi:10.1137/1.9780898717716.

**41** Mariano M. Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In *Computer Safety, Reliability, and Security (SAFECOMP)*, 2017. doi:10.1007/978-3-319-66266-4_14.

42 Andres Nötzli and Fraser Brown. LifeJacket: Verifying Precise Floating-Point Optimizations in LLVM. In *International Workshop on State Of the Art in Program Analysis (SOAP)*, 2016. doi:10.1145/2931021.2931024.

43 Scott Owens, Magnus O Myreen, Ramana Kumar, and Yong Kiam Tan. Functional Big-Step Semantics. In *European Symposium on Programming (ESOP)*, 2016. doi:10.1007/978-3-662-49498-1_23.

44 Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. Automatically Improving Accuracy for Floating Point Expressions. In *Conference on Programming Language Design and Implementation (PLDI)*, 2015. doi:10.1145/2737924.2737959.

45 A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1998. doi:10.1007/BFb0054170.

46 Johannes Aman Pohjola, Henrik Rostedt, and Magnus O. Myreen. Characteristic Formulae for Liveness Properties of Non-terminating CakeML Programs. In *Interactive Theorem Proving (ITP)*, 2019. doi:10.4230/LIPIcs.ITP.2019.32.

47 Markus Püschel, José M F Moura, Bryan Singer, Jianxin Xiong, Jeremy R Johnson, David A Padua, Manuela M Veloso, and Robert W Johnson. Spiral - A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. *International Journal of High Performance Computing Applications*, 18(1), 2004.

48 Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. A Unified Coq Framework for Verifying C Programs with Floating-Point Computations. In *Certified Programs and Proofs (CPP)*, 2016. doi:10.1145/2854065.2854066.

49 Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at Large: A Survey of Engineering of Formally Verified Software. *Foundations and Trends in Programming Languages*, 5, 2019. doi:10.1561/2500000045.

50 Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning Assistant for Floating-Point Precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013. doi:10.1145/2503210.2503296.

51 Hanan Samet. Proving the Correctness of Heuristically Optimized Code. *Communications of the ACM*, 21(7), 1978. doi:10.1145/359545.359569.

52 Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Optimization of Floating-point Programs with Tunable Precision. In *Programming Language Design and Implementation (PLDI)*, 2014. doi:10.1145/2594291.2594302.

53 Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics (TPHOL)*, 2008. doi:10.1007/978-3-540-71067-7_6.

54 Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In *International Symposium on Formal Methods (FM)*, 2015. doi:10.1007/978-3-319-19249-9_33.

55 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The Verified CakeML Compiler Backend. *Journal of Functional Programming*, 29, 2019. doi:10.1017/S0956796818000229.

56 Laura Titolo, Marco A Feliú, Mariano Moscato, and César A Munoz. An Abstract Interpretation Framework for the Round-off Error Analysis of Floating-Point Programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2018. doi:10.1007/978-3-319-73721-8_24.

57 Linus Torvalds. What is acceptable for -ffast-math?, 2001. URL: https://gcc.gnu.org/legacy-ml/gcc/2001-07/msg02150.html.

58 Vadim Zaliva. *HELIX: From Math to Verified Code*. PhD thesis, Carnegie Mellon University, 2020.

59 Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Principles of Programming Languages (POPL)*, 2012. doi:10.1145/2103656.2103709.