A Decade of Progress in Verifying Compiler Optimizations: A Formal Methods Perspective

Anand Kumar Keshavan, July 2025

Abstract

Compilers are a foundational component of the software development toolchain, yet their complexity makes them a persistent source of subtle and critical bugs. An incorrect optimization can silently corrupt an otherwise correct program, invalidating upstream verification and testing efforts. This is particularly perilous for safety-critical systems where software correctness is paramount. Over the past decade, the field of compiler verification has matured significantly, moving from a niche academic pursuit to a domain with practical, industrially relevant solutions. This survey provides a comprehensive review of the advancements in proving the correctness of code generated by compiler optimizers over the last 10 years. We chart the progress along two principal, and sometimes convergent, paths: whole-compiler verification, exemplified by the landmark CompCert project, which proves the compiler correct once and for all; and translation validation, a pragmatic post-hoc technique that verifies each compilation run and has been successfully applied to industrial compilers like LLVM and GCC. We present a detailed taxonomy of verified optimizations, covering foundational static analyses, scalar and global transformations, complex memory and loop optimizations, and the emerging challenges of verifying transformations for concurrency, weak memory models, and floating-point arithmetic. The survey concludes by examining the open challenges and future directions that will shape the next decade of research, including the verification of machine learning-based optimizations and the critical need for compositional correctness in an increasingly multi-language software ecosystem.

1. Introduction

The Enduring Challenge of Compiler Correctness

Compilers are the bedrock upon which modern software is built. They perform the

critical task of translating high-level, human-readable source code into low-level machine code that can be executed by a processor. This translation process is not merely a direct transcription; modern optimizing compilers perform a series of sophisticated and intricate program transformations to improve performance, reduce code size, or lower energy consumption.[1] While these optimizations are indispensable for achieving the performance required by contemporary applications, their complexity makes the compiler itself a large and error-prone piece of software.[2]

A bug in an optimizer can lead to a *miscompilation*, where a semantically correct source program is silently translated into an executable that behaves incorrectly.[3] Such errors are notoriously difficult to detect and debug because they defy the programmer's mental model of the source code and are often exposed only under specific runtime conditions.[3] The history of proving compiler correctness is nearly as old as the field of compilation itself, with the first such proof dating back to McCarthy and Painter in 1967 for the compilation of arithmetic expressions.[3] However, for decades, this remained a largely theoretical exercise, confined to small languages or individual transformations. The last decade, however, has witnessed a profound shift, transforming compiler verification into a field with tangible, practical, and increasingly indispensable results.

The High Stakes of Miscompilation in Critical Systems

For low-assurance software validated primarily by testing, the impact of compiler bugs is often mitigated, as testing is performed on the final executable and may expose the error.[3] The landscape changes dramatically for high-assurance and safety-critical systems, such as those in avionics, automotive, and medical devices. In these domains, validation by testing is insufficient and is increasingly complemented or replaced by formal methods like static analysis, model checking, and program proof.[2] These techniques provide mathematical guarantees about the behavior of the

source code. A bug in the compiler can shatter these guarantees, rendering the entire upstream verification effort worthless.[2] In this context, the compiler is no longer just a tool but a critical "weak link" in the chain of trust from specification to executable.[3]

The prevalence of miscompilation bugs in mature, widely-used compilers is well-documented. The Csmith study, for instance, used randomized differential testing to find hundreds of wrong-code bugs in production compilers like GCC and LLVM. The

same study found the formally verified CompCert C compiler to be remarkably resilient, finding no middle-end bugs over the course of six CPU-years of testing.[4] This stark contrast provides powerful empirical evidence for the value of formal verification in compiler construction.

An Overview of Modern Verification Paradigms

The advancements of the last decade have largely coalesced around two dominant methodologies for ensuring compiler correctness, each with its own philosophy and trade-offs.

The first is **whole-compiler verification**, an ambitious, proof-centric approach that aims to establish, once and for all, that the compiler's implementation is correct. This involves applying formal methods to the source code of the compiler itself, culminating in a machine-checked mathematical proof that it preserves the semantics of any program it compiles successfully. The flagship project in this area is CompCert, a formally verified C compiler that has served as both a proof of concept and a benchmark for the field.[3]

The second, more pragmatic, paradigm is **translation validation (TV)**. Rather than verifying the entire compiler, which may be infeasible for massive, rapidly evolving codebases like LLVM and GCC, translation validation verifies the result of each individual compilation run. After the compiler produces target code from a source program, an external tool—the validator—proves *a posteriori* that this specific translation was correct.[4]

The maturation of these two distinct approaches was not coincidental but rather the result of parallel, causally linked developments. The sustained success of the CompCert project provided the first concrete evidence that end-to-end verification of a realistic, optimizing compiler was not merely a theoretical possibility but an achievable engineering goal. This established a gold standard and catalyzed a wave of research in the area. Concurrently, the sheer scale and complexity of industrial compilers, often written in unsafe languages like C++, made the whole-compiler verification approach seem intractable for them.[12] This created a pressing need for a viable alternative. Translation validation filled this gap, but its practicality was contingent on the ability to automatically and efficiently solve the complex logical formulas—the verification conditions—that it generates. The dramatic rise in the

power, scope, and availability of automated reasoning tools, particularly Satisfiability Modulo Theories (SMT) solvers like Z3 [14], provided the crucial enabling technology that made translation validation practical at an industrial scale. Projects like Alive and Vellvm for LLVM are direct outgrowths of this synergy.[17] Thus, the contemporary landscape of compiler verification is shaped by this duality: the holistic, proof-centric model pioneered by CompCert, and the agile, validation-based model for industrial compilers, powered by modern SMT solvers.

Structure of this Survey

This survey charts the progress in the field over the last decade. Section 2 details the foundational methodologies of whole-compiler verification and translation validation. Section 3 presents an in-depth case study of the CompCert project, examining its architecture, proof methodology, and evolution. Section 4 explores the application of translation validation to the industrial compilers LLVM and GCC, focusing on the role of SMT solvers and specialized frameworks. Section 5 provides a detailed taxonomy of verified optimizations, assessing the maturity and challenges across various classes of program transformations. Finally, Section 6 looks to the future, discussing the open challenges of verifying machine learning-based optimizations and achieving compositional correctness for multi-language software.

2. Foundational Methodologies for Proving Correctness

At the heart of proving compiler correctness lies the concept of *semantic preservation*. The goal is to formally demonstrate that the compiled target code C behaves in a way that is consistent with the semantics of the source program S. The last decade has seen the refinement of three primary methodologies for establishing this property, each with a different trust model and set of practical trade-offs.

Whole-Compiler Verification: The Proof-Centric Approach

Whole-compiler verification is the most rigorous approach. Its principle is to formally prove, once and for all, that the compiler implementation itself is correct. This is captured by a theorem of the form:

$$\forall S, C, \text{Comp}(S) = \text{OK}(C) \Rightarrow S \approx C$$

where Comp is the compiler function, S is the source program, C is the compiled code, and S ≈ C is the semantic preservation relation.3 This proof is typically constructed using a proof assistant like Coq or Isabelle/HOL and is applied directly to the source code of the compiler's transformation passes.
The semantic preservation relation S ≈ C is subtle. A naive definition of equivalence, where S and C have identical sets of behaviors, is too strong. Compilers are permitted to make choices where the source language semantics are non-deterministic (e.g., expression evaluation order in C). Furthermore, a key purpose of optimization is to eliminate undesirable behaviors, such as "going wrong" states that correspond to undefined behavior (e.g., null pointer dereferencing).[3] The standard definition is therefore a

forward simulation (or refinement): if the source program S is safe (cannot go wrong), then any observable behavior of the compiled code C must be one of the possible observable behaviors of S.[3] For deterministic languages, which is the case for most of CompCert's languages, this simplifies to the property that if

S executes to produce a valid observable behavior B, then C must also execute to produce the same behavior B.[3]

In practice, verifying a monolithic compiler is intractable. The proof is therefore constructed compositionally. The compiler is structured as a sequence of passes, each transforming code from one intermediate language (IL) to the next. A semantic preservation theorem is proven for each pass. The end-to-end correctness of the entire compiler is then guaranteed by the transitivity of the simulation relation.[3]

Translation Validation: The Pragmatic, Post-Hoc Approach

In contrast to the *a priori* proof of whole-compiler verification, translation validation (TV) takes a post-hoc approach. Instead of verifying the compiler itself, it verifies the *result* of each compilation. After the compiler, which can be treated as a black or grey box, produces C from S, an external validator tool checks if the S ≈ C relation holds for

that specific pair of programs.[4]

This shifts the burden of trust from the large, complex, and often unverified compiler to the validator. For the strongest formal guarantees, the validator itself must be formally verified to be sound, meaning it never returns a false positive:

$$\forall S, C, \text{Validate}(S, C) = \text{true} \Rightarrow S \approx C$$

When using a verified validator, the overall verified system is a composite function that invokes the untrusted compiler and then checks its output with the trusted validator, failing with a compile-time error if the validation check fails.6
The primary advantage of TV is its practicality for large, rapidly evolving industrial compilers like LLVM and GCC. These compilers are written in unsafe languages (C++) and are far too large and complex for a full-scale proof effort.[12] TV allows verification to be retrofitted onto these existing systems without modifying their core. It is also particularly well-suited for complex optimizations based on heuristics (e.g., register allocation), where proving the algorithm correct is difficult, but checking that a given output is a valid instantiation of the transformation is much easier.[4]

Certifying Compilers and Proof-Carrying Code (PCC)

A certifying compiler is a related concept that bridges the gap between the producer and consumer of code. Instead of simply producing compiled code C, it also produces a machine-checkable *proof* or *certificate* π that C adheres to a predefined safety policy (e.g., type safety, memory safety).[3]

In the PCC model, the code consumer does not need to trust the compiler or the code producer. The consumer only needs a small, trusted proof checker to validate that the certificate π is a valid proof for the code C with respect to the safety policy.[3] This is a powerful model for ensuring the safety of mobile code in security-sensitive environments.

PCC can be viewed as a form of translation validation where the evidence for correctness is packaged with the code for later validation by a third party. A fully verified compiler can be seen as an ultimate certifying compiler, where the certificate is implicitly guaranteed by the compiler's own correctness proof.

Table 1 provides a comparative summary of these three foundational methodologies.

Table 1: Comparison of Compiler Correctness Methodologies

Methodology	Core Principle	Strengths	Weaknesses	Key Examples/Tools
Whole-Compiler Verification	Prove the compiler correct once, for all inputs.	Strongest formal guarantee; no runtime overhead for validation.	Extremely high proof effort; difficult to maintain and extend the compiler.	CompCert, CakeML
Translation Validation	Prove each individual compilation correct after it runs.	Practical for complex, legacy compilers (LLVM, GCC); can handle heuristics; can be developed independently.	Incomplete (may have false negatives); validator must be trusted or separately verified.	Alive, Vellvm, TVOC
Certifying Compilers / PCC	Bundle compiled code with a machine-checkable proof of safety.	Minimal trusted computing base for the code consumer; ideal for mobile code and security.	Proofs can be large; typically targets safety policies, not full semantic equivalence.	Touchstone compiler, early PCC research

3. Case Study: The CompCert Formally Verified C Compiler

The CompCert project stands as the most significant and successful demonstration of the whole-compiler verification approach. Initiated by Xavier Leroy and developed over more than 15 years, it has evolved from a groundbreaking research project into a robust, commercially available tool used in safety-critical industries.[5] Its design and evolution provide crucial lessons in the engineering of large-scale verified software.

Architectural Overview: A Pipeline of Verified Passes

At its core, CompCert's design is a testament to the principle of "divide and conquer" as a proof engineering strategy. Verifying a direct translation from C to assembly would be an insurmountable task due to the enormous semantic gap between the two languages.[3] Instead, CompCert decomposes the compilation process into a pipeline of approximately 20 distinct passes that operate over a series of 11 intermediate languages (ILs).[5] This architecture is not merely for implementation convenience; it is fundamental to making the verification effort tractable.

The compilation pipeline begins with a large subset of the C language (often referred to as CompCert C or Clight) and progressively lowers the level of abstraction. Each IL is designed to make a specific feature of the machine or a particular aspect of the computation explicit.[6] Key stages in this pipeline include:

- **Clight:** A simplified subset of C where side effects within expressions have been pulled out into separate statements, and the evaluation order is deterministic. This serves as the well-defined source language for the verified backend.[23]
- **Cminor:** A low-level, structured imperative language that makes memory loads/stores and stack allocation explicit. Variables whose addresses are not taken can be targeted for register allocation.[6]
- **RTL (Register Transfer Language):** A classic compiler representation using a control-flow graph (CFG) of instructions that operate on an infinite set of pseudo-registers. This representation is the natural domain for standard dataflow analyses and optimizations like constant propagation and common subexpression elimination, as well as for register allocation.[3]
- **Mach and Asm:** The final stages of the pipeline, which map the program onto the concrete architecture, dealing with activation record layout, calling conventions, and the specific instruction set of the target processor (e.g., PowerPC, ARM, RISC-V, x86).[9]

By breaking the compilation down into many small, focused steps, the proof of correctness for the entire compiler is decomposed into a series of smaller, more manageable proofs, one for each pass. Each proof need only bridge the small semantic gap between its input and output ILs.

The Role of the Coq Proof Assistant in Implementation and Proof

CompCert is inextricably linked with the Coq proof assistant. Coq is used not only to formalize the semantics of all languages and to conduct the correctness proofs but also to implement the compiler passes themselves.[3] The transformation algorithms are written as functions in Coq's purely functional programming language, Gallina. This approach has a profound advantage: because the implementation is a first-class object within Coq's logic, it can be reasoned about directly. There is no gap between the algorithm that is proven correct and the code that runs, a common source of error in verification projects that prove properties about an abstract model of an implementation written separately in another language.[3]

Once the functional implementation of the compiler is proven correct within Coq, Coq's extraction facility is used to automatically generate executable OCaml code. This extracted code is then combined with unverified components (like the parser and linker) and compiled by a standard OCaml compiler to produce the final ccomp executable.[3] This verified development workflow, from a formal specification in Coq directly to an executable binary, provides an exceptionally high degree of assurance. The proof effort is substantial, with the Coq proofs being many times larger than the code they verify.[3]

Semantic Preservation: The Core Correctness Theorem

The central guarantee provided by CompCert is a semantic preservation theorem, formally proven in Coq. This theorem states that the observable behavior of the compiled assembly code is a faithful refinement of the observable behavior of the source C program.[3] Observable behavior includes termination, non-termination (divergence), "going wrong" on an undefined operation, and the trace of input/output system calls made by the program.[3]

The end-to-end proof is achieved by composing the individual proofs for each pass. Each pass is proven correct using a *forward simulation* argument. This involves defining a relation ~ between states of the source and target ILs for that pass and proving that for every step of execution in the source program, there is a corresponding sequence of one or more steps in the target program that preserves the simulation relation and produces the same observable events.[3] For example, the

proof for register allocation shows that for every step in the RTL program, the LTL program can take a corresponding step, preserving an invariant that relates the values in RTL pseudo-registers to their assigned locations (hardware registers or stack slots) for all

live variables.[3] The formal semantics for all languages are defined within Coq, having evolved from initial big-step operational semantics to more expressive small-step semantics that can model non-terminating programs.[24]

A Decade of Evolution: From Research Project to Industrial Tool

Over the past 10 years, CompCert has matured from a proof-of-concept research project into a feature-rich, robust compiler suitable for industrial use in critical domains. Its release history tracks a steady expansion of capabilities [9]:

- **Language Coverage:** The supported subset of C has grown to encompass nearly all of ISO C 2011. This includes notoriously difficult-to-formalize features like goto statements, 64-bit integers (long long), volatile memory accesses for interacting with hardware, bit-fields in structs, and even unstructured switch statements (e.g., Duff's device).[9]
- **Target Architectures:** Initially targeting only PowerPC, CompCert now generates efficient code for the most common embedded and server architectures, including ARM (32-bit and 64-bit AArch64), x86 (32-bit and 64-bit), and RISC-V (32-bit and 64-bit).[9]
- **Optimizations and Performance:** While its primary goal is correctness, not aggressive optimization, CompCert implements a solid suite of standard dataflow-based optimizations, including constant propagation, common subexpression elimination, and dead code elimination, along with an effective graph-coloring register allocator and, more recently, function inlining.[24] Its performance is generally competitive with GCC at the -O1 optimization level, making it a viable choice for performance-sensitive embedded applications.[3]
- **Industrial Adoption and Recognition:** CompCert is commercially distributed by AbsInt GmbH for use in safety-critical industries like avionics and automotive software.[5] Its impact has been recognized with prestigious honors, including the 2021 ACM Software System Award, affirming its status as one of the most significant software systems to emerge from academic research.[9]

Despite its success, CompCert is not without limitations. The trust story has well-defined boundaries: the C preprocessor, parser, assembler, and linker are external, unverified tools.[3] Furthermore, research on extending CompCert to handle separate compilation uncovered a subtle bug related to the interaction between

const and extern variables, a powerful reminder that even formal specifications can have flaws and that compositional reasoning introduces new challenges.[23]

4. Translation Validation for Industrial Compilers: LLVM and GCC

While CompCert demonstrates the feasibility of building a verified compiler from the ground up, this approach is generally considered intractable for existing industrial compilers like LLVM and GCC. These systems are massive, comprising millions of lines of C++, evolve at a rapid pace with contributions from hundreds of developers, and were not designed with formal verification in mind.[12] For these critical pieces of infrastructure, translation validation (TV) has emerged as the only practical path toward formal correctness guarantees.[10]

Bridging the Gap: Applying Formal Methods to Legacy Codebases

The core challenge in applying TV to industrial compilers is to retrofit formal analysis onto a system that was not built for it. The goal is to treat the compiler's optimization passes largely as black boxes, taking the intermediate representation (IR) before a pass and the IR after the pass and proving their semantic equivalence without needing to reason about the implementation of the pass itself. This requires a formal semantics for the compiler's IR and a powerful engine for automatically proving the equivalence theorems that arise. The design of a compiler's IR has a profound impact on the feasibility of this endeavor. The LLVM project, with its well-defined, typed, SSA-based IR, has proven to be a significantly more amenable target for formal analysis than the older, more complex representations of compilers like GCC. This has led to LLVM becoming the de facto platform for cutting-edge research in translation validation, creating a feedback loop where better tools are built for LLVM, which in

turn encourages more verification research focused on it.

SMT-Based Equivalence Checking and Symbolic Execution

The engine that powers most modern translation validation systems is the Satisfiability Modulo Theories (SMT) solver.[14] The problem of proving the equivalence of a source program fragment

S and a target fragment T is typically transformed into a satisfiability problem. The most common technique is to construct a logical formula that is satisfiable only if there exists an input for which S and T produce different results. This formula is then passed to an SMT solver. If the solver returns UNSAT (unsatisfiable), it constitutes a proof that no such input exists, and therefore S and T are equivalent for all inputs.[16]

Symbolic execution is a key technique for automatically generating these logical formulas from program code.[2] Instead of executing a program with concrete data, symbolic execution uses symbolic variables to represent arbitrary inputs. As the program executes, the analysis tracks the symbolic state and accumulates path conditions—constraints on the input variables that must hold to follow a particular execution path. The final symbolic state at the end of a path represents the program's output as a formula over its inputs. By generating such formulas for both the source and target programs, a TV system can construct the necessary equivalence query for the SMT solver.

Specialized Frameworks: Verifying LLVM with Vellvm and Alive

The relative clarity and structure of the LLVM IR have enabled the creation of powerful, specialized frameworks for its verification.

- **Vellvm:** This project provides a formalization of the LLVM IR, its type system, and its SSA properties within the Coq proof assistant.[18] Vellvm serves as a foundational framework for building verified LLVM passes. Its most notable achievement is the formal verification of a variant of the mem2reg pass, a crucial transformation that promotes memory accesses to SSA registers. This verification is particularly challenging because SSA properties are

inherently non-local, depending on the entire control-flow graph of a function.[18] Vellvm enables the extraction of verified passes that can be plugged directly into the LLVM toolchain.

- **Alive:** This framework provides a domain-specific language (DSL) for specifying and automatically verifying peephole optimizations, such as those found in LLVM's InstCombine pass.[17] An optimization is written as a source pattern, a target pattern, and an optional precondition. Alive translates this specification into an SMT query that checks for counterexamples. It is designed to handle the subtleties of LLVM's semantics, including
undef and poison values, which are used to signal undefined behavior and enable aggressive optimizations. Alive has been highly successful, finding dozens of bugs in LLVM's optimizer and is now used by LLVM developers to validate new optimizations.[17]
- **LifeJacket:** Recognizing the unique challenges of floating-point arithmetic (e.g., non-associativity, NaNs, signed zeros), LifeJacket extends Alive with support for verifying floating-point optimizations. It uses the SMT theory of floating-point numbers to accurately model IEEE 754 semantics and has uncovered several subtle bugs in LLVM's handling of floating-point code.[30]

Other efforts in validating LLVM have focused on specific, complex passes like register allocation, using dataflow analysis to infer the mapping between virtual registers and physical locations and proving its correctness.[11]

Approaches to Validating GCC's Register Transfer Language (RTL) Passes

Applying translation validation to the GNU Compiler Collection (GCC) presents a different set of challenges, largely due to the nature of its primary intermediate representation, RTL. RTL is an older, Lisp-like representation that is less structured than LLVM IR.[32] Research in this area has focused on building TV infrastructure capable of parsing and reasoning about RTL. These systems typically use GCC plugins to extract the RTL representation of a function before and after an optimization pass.[32] The core validation technique remains SMT-based, requiring the development of a formal model of RTL semantics within an SMT solver like Z3. For the many GCC passes that preserve the basic block structure of the program, a block-by-block equivalence check can be performed, significantly simplifying the proof obligation.[32]

5. A Taxonomy of Verified Optimizations

The past decade has seen the application of formal verification techniques to a wide array of compiler optimizations, ranging from simple local rewrites to complex, whole-program transformations. The maturity of verification varies significantly across these different classes of optimization, reflecting their inherent complexity and the suitability of existing formal methods. Table 2 provides a high-level overview of the state of the art.

Table 2: Status of Verification for Key Optimization Classes

Optimization Class	Verification Maturity	Dominant Technique	Key Challenges	Representative Work
Scalar/Peephole	Mature	SMT-based Translation Validation	Handling undefined behavior (undef, poison)	Alive [17], Peek [33]
Dataflow/SSA	Developing	Simulation Proofs, TV	Non-local properties of SSA form	Vellvm (mem2reg) [29], CompCert (CSE) [24]
Register Allocation	Developing	Verified Implementation, TV	Combinatorial complexity, heuristics	CompCert (IRC) [26], LLVM TV [31]
Alias Analysis	Developing	Abstract Interpretation	Soundly abstracting memory state	CompCert (Points-to) [34]
Loop Transformations	Active Research	Polyhedral Model, Specialized TV	Drastic structural changes to code	Polyhedral CodeGen [35], TVOC [36]
Concurrency/	Active Research	Model Checking,	State-space	Valid-C [37],

Weak Memory		Specialized TV	explosion, complex semantics	Fairness Models [38]
Floating-Point	Active Research	SMT-based TV (FP Theory)	Non-algebraic properties, special values	LifeJacket [30], Icing [39]

Verifying Foundational Analyses: Dataflow, Alias, and Liveness Analysis

Correct optimizations are built upon the foundation of sound static analyses. Consequently, a critical step in verifying an optimizer is to first verify the correctness of the analyses that provide it with information about the program.

- **Dataflow Analysis:** The correctness of dataflow analyses, which propagate facts along a program's control-flow graph, is typically established within the framework of *abstract interpretation*.[2] This involves defining an abstract domain that safely approximates the concrete program state and proving that the analysis's transfer functions and fixed-point computation are sound with respect to the concrete program semantics.[41]
- **Alias Analysis:** Alias analysis, which determines whether different pointers can refer to the same memory location, is fundamental for any optimization involving memory. A landmark achievement in this area is the formal verification of a flow-sensitive, field-sensitive, intraprocedural points-to analysis for CompCert.[34] The proof, conducted in Coq, uses abstract interpretation to show that the analysis's abstract partitioning of memory is a sound over-approximation of all possible concrete memory layouts. Proving the soundness of such analyses is crucial, as an incorrect alias analysis can lead an optimizer to perform unsafe code transformations.[43]

Local Transformations: Peephole and Scalar Optimizations

This class of optimizations, which involves replacing small, local instruction patterns with more efficient ones, is one of the most mature areas for verification. The primary technique is translation validation powered by SMT solvers. The Alive framework for

LLVM is the leading example, providing a DSL to specify rewrite rules and automatically checking their validity against LLVM's semantics, including its complex rules for undefined behavior.[17] For verified compilers like CompCert, frameworks like Peek have been developed to verify peephole optimizations on the x86 backend. Peek cleverly separates the proof of local correctness for each rule from a single, global proof that any locally-correct rewrite preserves the overall program semantics, thereby simplifying the verification effort for individual optimizations.[33]

Global Transformations: SSA-based and Dataflow Optimizations

Global optimizations like Common Subexpression Elimination (CSE) and constant propagation operate over the entire control-flow graph of a function. In verified compilers like CompCert, these are proven correct using forward simulation proofs. The proof for CSE, for example, relies on a verified value analysis (a form of dataflow analysis) to identify redundant computations.[24] For SSA-based compilers like LLVM, verifying these transformations is more complex because the Static Single Assignment (SSA) form introduces non-local dependencies. The verification of LLVM's

mem2reg pass in the Vellvm framework represents a major milestone, requiring sophisticated reasoning about the dominance properties inherent to the SSA form.[29]

The Complexity of Memory: Register Allocation and Coalescing

Register allocation is a notoriously complex optimization. Because the underlying graph coloring problem is NP-hard, compilers employ a suite of sophisticated and aggressive heuristics to map an unbounded number of virtual registers to a finite set of physical machine registers.[45] Verifying these heuristic-driven algorithms directly is extremely challenging.

- **CompCert's dual approach:** CompCert's journey with register allocation is illustrative. Initially, it used an unverified, external implementation of the Iterated Register Coalescing (IRC) algorithm and employed translation validation to check the correctness of each allocation it produced.[26] This pragmatic approach provided the full correctness guarantee for the compiler without requiring a proof of the complex allocator itself. More recent work has achieved the gold standard:

a complete, purely functional implementation of the IRC algorithm has been written and proven totally correct (including termination) within Coq. This verified allocator is now integrated into CompCert.[26]

- **Translation validation for LLVM:** For LLVM, translation validation is the method of choice. This involves developing an external validator that can infer the complex mapping between virtual registers and their physical locations (which may be a register for part of a variable's lifetime and a stack slot for another, due to live-range splitting). The validator uses dataflow analysis to generate verification conditions that are then discharged by a proof checker, confirming that the allocation preserves the program's semantics.[11]

Transforming Control Flow: Loop Optimizations and the Polyhedral Model

Loop optimizations, such as fusion, tiling, and interchange, are essential for high-performance computing but are among the most difficult to verify because they radically restructure the program's control flow, invalidating simple state-machine simulation proofs.[36] The dominant formal framework for reasoning about these transformations is the

polyhedral model. This model represents nested loops as geometric objects (polyhedra) in an integer vector space, and transformations are expressed as affine transformations on these objects.[35] This provides a powerful mathematical foundation for proving correctness.

Recent research has made significant progress in formally verifying the code generation phase of a polyhedral optimizer—the part that translates the abstract polyhedral representation back into concrete, imperative loops. This work, done in Coq, involves proving that the generated loops correctly enumerate all and only the integer points within the transformed iteration domain.[35] Other approaches use translation validation, developing specialized "meta-rules" to handle reordering transformations where a one-to-one mapping between source and target states does not exist.[36]

Handling Modern Architectures: Concurrency, Weak Memory, and Floating-Point Arithmetic

These areas represent the cutting edge of compiler verification research, driven by the complexities of modern hardware and programming language standards.

- **Concurrency:** Verifying compiler optimizations for concurrent programs is exceptionally challenging. Many standard sequential optimizations, such as reordering or eliminating memory accesses, are unsound in the presence of threads interacting under a weak memory model.[37] Research in this area focuses on building validators that check whether a transformation is permissible under the formal memory models of languages like C11 and LLVM. This work is highly practical and has already uncovered several concurrency-related miscompilation bugs in LLVM.[37]
- **Weak Memory Models:** The formal semantics of weak memory models, which permit surprising out-of-order execution behaviors, are themselves a complex and active area of research. Verifying compiler optimizations requires reasoning about these intricate models. Recent work has focused on defining formal notions of fairness for these models and proving that compiler transformations do not violate liveness properties of concurrent programs.[38]
- **Floating-Point Arithmetic:** Floating-point (FP) arithmetic deviates significantly from real-number arithmetic; it is not associative, and it includes special values like Not-a-Number (NaN), signed zeros, and infinities. These properties invalidate many common algebraic simplifications. Verifying FP optimizations requires a precise formal model of the IEEE 754 standard. Frameworks like LifeJacket extend SMT-based translation validation to this domain by leveraging the SMT-LIB theories for floating-point numbers and bit-vectors, enabling the automatic verification of precise FP optimizations in LLVM.[30]

6. Future Directions and Open Challenges

While the last decade has seen compiler verification mature into a practical discipline, the field's trajectory is increasingly shaped by disruptive trends in the broader landscape of computing. The most significant future challenges lie at the intersection of verification and these external forces: the rise of artificial intelligence, the shift to component-based software, and the ever-increasing complexity of hardware.

The New Frontier: Verifying Machine Learning-Based Optimizations

One of the most significant recent trends in compiler design is the use of machine learning (ML) to replace or augment handcrafted heuristics. ML models are being used for complex optimization tasks such as phase ordering, inlining decisions, register allocation, and thread mapping.[52] These models can often discover better optimization strategies than human experts, but they introduce a formidable verification challenge.

The core problem is that the optimization policy is no longer an explicit, human-written algorithm but a "black box" model, such as a neural network, whose decision-making process is not readily interpretable.[53] Traditional verification techniques, which reason about the structure of algorithms, are inapplicable. How can one prove that a compiler driven by a learned model will never miscompile? This remains a critical open problem.

Several potential research directions are emerging. One pragmatic approach is to decouple correctness from performance. The ML model can be used to *propose* an optimization strategy from a set of pre-defined, individually verified transformations. In this model, even if the ML component makes a suboptimal performance choice, it can never cause a miscompilation because the building blocks it uses are already proven correct.[59] A second approach is to apply translation validation to the output of the ML-driven compiler. This treats the entire ML-compiler system as a black box and validates its output post-hoc. While this can catch errors, it does not provide any guarantees about the model itself. Formally verifying the properties of the ML models directly is a grand challenge that will require deep connections between the formal methods, machine learning, and compiler communities.

Beyond Monolithic Programs: Compositional Correctness for Multi-Language Software

The vast majority of verified compilers, including CompCert, are proven correct under a "whole-program" assumption: they compile a single, self-contained source program into a complete executable.[60] However, this model does not reflect the reality of

modern software development, which is increasingly component-based and polyglot. Systems are often constructed by linking together libraries and components written in different languages (e.g., a Python application using a C-based numerical library) and compiled with different toolchains.[60]

This reality exposes a critical gap in existing correctness theorems. The key open problem is **compositional compiler correctness**: proving that a compiled component is correct in a way that guarantees it can be safely linked with arbitrary other target components, including those that are unverified, written in unsafe languages, or compiled by different compilers.[60] This requires a correctness theorem that accounts for language interoperability.

A promising avenue of research is the design of a common, low-level target language that can safely mediate this interoperability. One such proposal is a **gradually type-safe target language** based on LLVM.[60] Such a language would support a spectrum of safety guarantees, from statically verified type-safe components (e.g., compiled from ML or Rust) to dynamically checked components and completely unsafe components (e.g., compiled from C). Safety at the boundaries between these components would be enforced by a combination of static typing and automatically inserted runtime checks or contracts. Projects like "The Next 700 Verified Compilers" aim to accelerate progress in this area by developing reusable, verified compiler building blocks that can be composed to build new, compositionally-correct compilers for a wide range of source languages.[61]

Revisiting the Grand Challenge: The Road to a Fully Verifying Compiler

In 2003, Sir Tony Hoare proposed the "Verifying Compiler" as a grand challenge for computing research.[62] His vision went beyond the semantic preservation provided by compilers like CompCert. A true verifying compiler would not only correctly translate a program but would also use mathematical and logical reasoning to help prove that the source program is correct with respect to a high-level functional specification provided by the programmer.[63]

This grand challenge requires a deep integration of program provers, advanced static analyzers, and automated theorem provers into the compiler toolchain. While this ultimate vision remains a distant goal for mainstream languages, the work of the last decade has laid an essential foundation. A verified compiler like CompCert provides

the crucial guarantee that the properties proven at the source level will not be invalidated by the compilation process. The future of the field lies in tightening this integration, creating toolchains where source-level verification and verified compilation work in concert to provide end-to-end, formally-backed guarantees of software correctness from specification all the way down to the executable.

7. Conclusion

The past decade has been a transformative period for the field of proving compiler correctness. What was once a domain of theoretical exploration has produced tangible artifacts and practical methodologies that are beginning to have a real-world impact on software reliability. The advancements can be synthesized into two major narratives. First, the maturation of the **CompCert project** has provided an existence proof that whole-compiler verification for a realistic, non-trivial language is achievable. It has evolved into an industrial-strength tool, setting a high bar for formal assurance and serving as an invaluable research platform. Second, the rise of powerful **SMT solvers** has enabled the success of **translation validation** as a viable, pragmatic approach for ensuring the correctness of complex, rapidly-evolving industrial compilers like LLVM and GCC. Frameworks like Alive have demonstrated that this post-hoc validation technique can be highly effective at finding deep bugs in heavily-used optimizers.

This progress has led to significant strides in verifying a wide spectrum of optimizations. The verification of local, peephole optimizations is now largely automated, and robust techniques have been developed for foundational dataflow analyses and complex transformations like register allocation. Active research continues to push the frontiers, tackling the immense challenges posed by loop transformations, concurrency, and the subtleties of floating-point arithmetic.

Despite this progress, a fundamental trade-off persists between the complete, ironclad guarantees of whole-compiler verification and the agile, scalable, but incomplete nature of translation validation. Looking forward, the field faces a new set of grand challenges that will define the research landscape for the next decade. The increasing adoption of machine learning in compiler heuristics presents a profound verification problem, as the logic of optimization becomes opaque and learned rather than explicitly programmed. Simultaneously, the shift towards polyglot,

component-based software engineering demands a move beyond whole-program verification to a new theory of compositional compiler correctness that can guarantee safe interoperability. The trajectory of the field has clearly shifted. The central question is no longer "Can we verify a compiler?" but rather, "How do we verify the compilers people actually use, for the software they actually write, on the hardware they actually run?" Answering this question will require continued innovation and a deep, collaborative effort between the formal methods, programming languages, and compiler communities.

Works cited

1. Options That Control Optimization - Using the GNU Compiler Collection (GCC), accessed on July 5, 2025, https://gcc.gnu.org/onlinedocs/gcc-3.2.3/gcc/Optimize-Options.html
2. A Survey of Automated Techniques for Formal Software Verification, accessed on July 5, 2025, https://cse.usf.edu/~haozheng/lib/verification/general/survey-sw-fv.pdf
3. Formal verification of a realistic compiler - Xavier Leroy, accessed on July 7, 2025, https://xavierleroy.org/publi/compcert-CACM.pdf
4. Progress in Formal Verification of Compilers: A Survey, accessed on July 10, 2025, https://web.cecs.pdx.edu/~apt/cs510comp/CompilerVerification.pdf
5. CompCert – A Formally Verified Optimizing Compiler - Xavier Leroy, accessed on July 7, 2025, https://xavierleroy.org/publi/erts2016_compcert.pdf
6. CS 6120: Formal Verification of a Realistic Compiler - Computer Science Cornell, accessed on July 15, 2025, https://www.cs.cornell.edu/courses/cs6120/2022sp/blog/compcert/
7. Survey of Existing Tools for Formal Verification - OSTI, accessed on July 16, 2025, https://www.osti.gov/servlets/purl/1166644
8. Software correctness from first principles | Formal Land, accessed on July 14, 2025, https://formal.land/blog/2024/06/05/software-correctness-from-first-principles
9. CompCert - Main page, accessed on July 26, 2025, https://compcert.org/
10. Translation Validation of Optimizing Compilers - NYU Computer ..., accessed on July 18, 2025, https://cs.nyu.edu/media/publications/fang_yi.pdf
11. Translation validation for compilation verification - IDEALS, accessed on July 8, 2025, https://www.ideals.illinois.edu/items/118300
12. Some Goals for High-impact Verified Compiler Research - Embedded in Academia, accessed on July 12, 2025, https://blog.regehr.org/archives/1565
13. Formal Verification of Translation Validators - Jean-Baptiste Tristan, accessed on July 19, 2025, https://jtristan.github.io/papers/popl08.pdf
14. Accurate and Extensible Symbolic Execution of Binary Code based on Formal ISA SemanticsThis work was supported by the German Federal Ministry of Education and Research (BMBF) within projects Scale4Edge under grant no. 16ME0127, ECXL under grant no. 01IW22002, and VE-HEP under grant no. 16KIS1342. - arXiv,

accessed on July 20, 2025, https://arxiv.org/html/2404.04132v2
15. Satisfiability modulo theories - Wikipedia, accessed on July 8, 2025, https://en.wikipedia.org/wiki/Satisfiability_modulo_theories
16. A gentle introduction to SMT-based program analysis | Fura Labs, accessed on July 26, 2025, https://furalabs.com/blog/2023/02/12/intro_to_smt_analysis
17. Practical Verification of Peephole Optimizations with Alive - Rutgers ..., accessed on July 26, 2025, https://people.cs.rutgers.edu/~santosh.nagarakatte/papers/cacm-feb-2018.pdf
18. Vellvm: Verified LLVM - CIS UPenn, accessed on July 26, 2025, https://www.cis.upenn.edu/~stevez/vellvm/
19. Formal Verification of a Realistic Compiler - ResearchGate, accessed on July 26, 2025, https://www.researchgate.net/publication/37794752_Formal_Verification_of_a_Realistic_Compiler
20. a trustworthy compiler - CompCert C, accessed on July 26, 2025, https://compcert.org/man/manual001.html
21. The Design and Implementation of a Certifying Compiler - CMU School of Computer Science, accessed on July 26, 2025, https://www.cs.cmu.edu/afs/cs/academic/class/15745-s06/web/handouts/necula-pldi98.pdf
22. Open Problems with Certifying Compilation - USENIX, accessed on July 26, 2025, https://www.usenix.org/conference/14th-usenix-security-symposium/open-problems-certifying-compilation
23. CS 6120: CompCert: the Double-Edged Sword of Verification - Computer Science Cornell, accessed on July 26, 2025, https://www.cs.cornell.edu/courses/cs6120/2020fa/blog/compcert/
24. Research objectives - CompCert, accessed on July 26, 2025, https://compcert.org/research.html
25. The CompCert verified compiler, accessed on July 26, 2025, https://compcert.org/doc/
26. Formal Verification of Coalescing Graph-Coloring ... - cs.Princeton, accessed on July 26, 2025, https://www.cs.princeton.edu/~appel/papers/regalloc.pdf
27. Chapter 2 Installation instructions - CompCert, accessed on July 26, 2025, https://compcert.org/man/manual002.html
28. Formalizing the LLVM Intermediate Representation for Verified Program Transformations - University of Pennsylvania, accessed on July 26, 2025, https://repository.upenn.edu/bitstreams/56441981-63fb-410e-a540-16a1d899ead0/download
29. Formal Verification of SSA-Based Optimizations for LLVM, accessed on July 26, 2025, https://www.cs.rutgers.edu/~santosh.nagarakatte/pldi2013.pdf
30. LifeJacket: Verifying Precise Floating-Point Optimizations in LLVM, accessed on July 26, 2025, https://homes.cs.washington.edu/~ztatlock/599z-17sp/papers/lifejacket-notzli-soap16.pdf
31. A Translation Validation Algorithm for LLVM Register ... - IDEALS, accessed on July

26, 2025, https://www.ideals.illinois.edu/items/120620/bitstreams/395802/data.pdf
32. Towards validation of RTL passes of the GCC ... - Eashan Gupta, accessed on July 26, 2025, https://eash3010.github.io/docs/BTP_report.pdf
33. Verified Peephole Optimizations for CompCert - University of ..., accessed on July 26, 2025, https://homes.cs.washington.edu/~ztatlock/pubs/peek-mullen-pldi16.pdf
34. (PDF) A Formally-Verified Alias Analysis - ResearchGate, accessed on July 26, 2025, https://www.researchgate.net/publication/234027199_A_Formally-Verified_Alias_Analysis
35. Verified Code Generation for the Polyhedral Model - Xavier Leroy, accessed on July 26, 2025, https://xavierleroy.org/publi/polyhedral-codegen.pdf
36. Translation Validation of Loop Optimizations and Software Pipelining in the TVOC Framework In memory of Amir Pnueli - NYU Computer Science, accessed on July 26, 2025, https://cs.nyu.edu/~goldberg/pubs/bg_pipelining10.pdf
37. Validating Optimizations of Concurrent C/C++ Programs, accessed on July 26, 2025, https://plv.mpi-sws.org/validc/paper.pdf
38. Making Weak Memory Models Fair, accessed on July 26, 2025, https://www.cs.tau.ac.il/~orilahav/papers/oopsla2021.pdf
39. Verified Compilation and Optimization of Floating-Point Programs in CakeML - DROPS, accessed on July 26, 2025, https://drops.dagstuhl.de/storage/00lipics/lipics-vol222-ecoop2022/LIPIcs.ECOOP.2022.1/LIPIcs.ECOOP.2022.1.pdf
40. Data-flow analysis - Wikipedia, accessed on July 26, 2025, https://en.wikipedia.org/wiki/Data-flow_analysis
41. Combining Model Checking and Data-Flow Analysis - SoSy-Lab, accessed on July 26, 2025, https://www.sosy-lab.org/research/pub/2018-HBMC.Combining_Model_Checking_and_Data-Flow_Analysis.pdf
42. Detecting Errors with Configurable Whole-program Dataflow Analysis - Manning College of Information & Computer Sciences, accessed on July 26, 2025, https://people.cs.umass.edu/~emery/pubs/detecting-errors.pdf
43. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages - LLVM, accessed on July 26, 2025, https://llvm.org/pubs/2006-05-12-PLDI-SAFECode.pdf
44. Alias Analysis for Assembly - CiteSeerX, accessed on July 26, 2025, https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=8720943e74c77bb262e1623fc8f712d0371a13d9
45. Register Allocation by Pseudo-Boolean Optimization - Harvard DASH, accessed on July 26, 2025, https://dash.harvard.edu/bitstream/handle/1/37364763/HORTON-SENIORTHESIS-2020.pdf?sequence=1&isAllowed=y
46. A Survey on Register Allocation - UCLA Compilers Group ·, accessed on July 26, 2025, http://compilers.cs.ucla.edu/fernando/publications/drafts/survey.pdf
47. Ensuring Correctness of Compiled Code - LLVM, accessed on July 26, 2025, https://llvm.org/pubs/2009-05-EnsuringCorrectnessOfCompiledCode.pdf

48. Loopy: Programmable and Formally Verified Loop ... - Kedar Namjoshi, accessed on July 26, 2025, https://kedar-namjoshi.github.io/papers/Namjoshi-Singhania-SAS-2016.pdf
49. Automatic Equivalence Checking of UF+IA Programs, accessed on July 26, 2025, https://web.ist.utl.pt/nuno.lopes/pubs/cork-spin13.pdf
50. Compositional relaxed concurrency - PMC - PubMed Central, accessed on July 26, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC5597728/
51. 152 Model Checking on Multi-execution Memory Models - Programming Languages & Verification, accessed on July 26, 2025, https://plv.mpi-sws.org/genmc/oopsla2022-wmc.pdf
52. (PDF) MILEPOST GCC: machine learning based research compiler, accessed on July 26, 2025, https://www.researchgate.net/publication/29633339_MILEPOST_GCC_machine_learning_based_research_compiler
53. Machine Learning in Compiler Optimization - University of Edinburgh Research Explorer, accessed on July 26, 2025, https://www.research.ed.ac.uk/files/70048258/Machine_Learning_in_compiler_optimisation.pdf
54. Machine-Learning-Based Self-Optimizing Compiler Heuristics - JKU ePUB, accessed on July 26, 2025, https://epub.jku.at/download/pdf/8305195.pdf
55. Compiler Optimization Parameter Selection Method Based on Ensemble Learning - MDPI, accessed on July 26, 2025, https://www.mdpi.com/2079-9292/11/15/2452
56. Machine Learning in Compiler Optimisation - arXiv, accessed on July 26, 2025, http://arxiv.org/pdf/1805.03441
57. A Machine Learning Based Compiler Optimization Technique - ResearchGate, accessed on July 26, 2025, https://www.researchgate.net/publication/382123249_Machine_Learning_Based_Compiler_Optimization_Technique
58. MLGO: A Machine Learning Framework for Compiler Optimization - Google Research, accessed on July 26, 2025, https://research.google/blog/mlgo-a-machine-learning-framework-for-compiler-optimization/
59. Machine Learning in Compiler Optimization - Reddit, accessed on July 26, 2025, https://www.reddit.com/r/Compilers/comments/ncmght/machine_learning_in_compiler_optimization/
60. Verified Compilers for a Multi-Language World - Northeastern ..., accessed on July 26, 2025, http://www.ccs.neu.edu/home/amal/papers/verifcomp.pdf
61. The next 700 verified compilers Research Project, 2022 – 2025, accessed on July 26, 2025, https://research.chalmers.se/en/project/10324
62. The Verifying Compiler, a Grand Challenge for Computing Research - ResearchGate, accessed on July 26, 2025, https://www.researchgate.net/publication/221550985_The_Verifying_Compiler_a_Grand_Challenge_for_Computing_Research
63. The Verifying Compiler: A Grand Challenge for Computing Research, accessed on July 26, 2025, https://www.csl.sri.com/users/shankar/GC04/hoare-compiler.pdf

64. The Verifying Compiler: A Grand Challenge for Computing Research | Request PDF, accessed on July 26, 2025, https://www.researchgate.net/publication/290959235_The_Verifying_Compiler_A_Grand_Challenge_for_Computing_Research
65. [GPCE] Language Design meets Verifying Compilers - YouTube, accessed on July 26, 2025, https://www.youtube.com/watch?v=fWSGhxyTG-4