Verified Peephole Optimizations for CompCert

Eric Mullen
University of Washington, USA
emullen@cs.washington.edu

Daryl Zuniga
University of Washington, USA
zunigad@cs.washington.edu

Zachary Tatlock
University of Washington, USA
ztatlock@cs.washington.edu

Dan Grossman
University of Washington, USA
djg@cs.washington.edu

Abstract

Transformations over assembly code are common in many compilers. These transformations are also some of the most bug-dense compiler components. Such bugs could be eliminated by formally verifying the compiler, but state-of-the-art formally verified compilers like CompCert do not support assembly-level program transformations. This paper presents Peek, a framework for expressing, verifying, and running meaning-preserving assembly-level program transformations in CompCert. Peek contributes four new components: a lower level semantics for CompCert x86 syntax, a liveness analysis, a library for expressing and verifying peephole optimizations, and a verified peephole optimization pass built into CompCert. Each of these is accompanied by a correctness proof in Coq against realistic assumptions about the calling convention and the system memory allocator.

Verifying peephole optimizations in Peek requires proving only a set of *local* properties, which we have proved are sufficient to ensure *global* transformation correctness. We have proven these local properties for 28 peephole transformations from the literature. We discuss the development of our new assembly semantics, liveness analysis, representation of program transformations, and execution engine; describe the verification challenges of each component; and detail techniques we applied to mitigate the proof burden.

DOI: http://dx.doi.org/10.1145/(to come)

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal Methods; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical Verification; D.3.4 [*Processors*]: Compilers

General Terms Languages, Verification

Keywords Formal Methods, Verification, Compilers, Coq, CompCert, Peek

1. Introduction

CompCert is a practical, fully formally verified C compiler with x86, PowerPC, and ARM back-ends. CompCert provides a machine checkable proof of semantic preservation for every compilation pass. Thus, if the source program is a legal C program and compilation succeeds, then the assembly produced has the same observable behavior as the source program. Previous work [9, 29] has repeatedly shown that CompCert is empirically more reliable than traditionally developed compilers, with multiple CPU-years of compiler fuzzing resulting in zero bugs discovered within the verified portions of CompCert.

Unfortunately, proving that every program transformation in CompCert preserves program meaning is incredibly difficult. CompCert's suite of optimizations has grown due to prior research [20, 21, 24–26], but still is not as extensive as other mature compilers. Adding optimizations is limited by the effort required to verify them. Optimizations that are often "relatively easy" to develop in a traditional, unverified compiler can be extremely challenging to prove correct in CompCert. This extra difficulty and how we mitigate it for *x86-assembly peephole optimizations in CompCert*, is the focus of our work.

Peephole Optimization Peephole optimization is a classic part of the compiler optimizer's toolkit [15, 22]. The approach scans an assembly program for a sequence of instructions matching a syntactic template and replaces it with a faster, equivalent sequence. Simple examples include re-

placing the move of 0 into a register with a self-xor operation, removing adjacent move instructions that together have no effect, or choosing different instructions to increase instruction-level parallelism (e.g., x86 has two ways to add registers that each use different functional units).

Peephole optimizations are particularly challenging for formal verification. They provide performance improvements unavailable via higher-level compiler intermediate representations because they can exploit low-level, architecture specific details and "clean up" pathologies of low-level code generation. They are challenging because the detailed semantics of assembly language includes subtle side effects on machine state. Peephole optimizations in unverified compilers typically make unstated assumptions about invariants that "happen to hold" in code generated by the compiler.

For example, as mentioned above, there are two ways to perform addition in x86, and peephole optimizations can replace one with the other. But such transformations require care, since the two forms of adding are not exactly equivalent: one may set a carry-flag register that the other never does. While other compilers may simply assume facts like "the flag register is live only immediately after a `cmp` instruction," we must verify a liveness analysis to ensure this detail. Due to such implicit assumptions, along with the inherent complexity of assembly semantics, peephole optimizations have been deceptively tricky in practice. For example, Lopes et al. [13] applied Csmith [29] to LLVM and found that peephole passes were the most bug-dense component of LLVM.

Assembly semantics in CompCert CompCert provides several carefully engineered intermediate program representations, where the lowest level is an assembly language. For each intermediate language, CompCert formally specifies a *dynamic semantics* and proves that transforming programs from one language to another preserves meaning. These semantics are engineered to balance the tension between proving correctness and supporting optimizations. Prior to our work, CompCert did not perform optimizations on x86 assembly, and thus the lowest-level language semantics were not engineered to support transformations.

As Section 5 explains in detail, CompCert's x86 semantics models an infinitely large memory with addresses (pointers) represented using mathematical integers. This model makes it impossible to prove many *correct* peephole optimizations, that is, many optimizations which would always preserve behavior on actual x86 hardware, because the semantics assumes, conservatively, that certain bit and arithmetic operations applied to pointers are undefined. To resolve this semantic mismatch, we defined a new, lower-level semantics that models pointers as 32-bit values and proved our semantics equivalent to the prior assembly semantics.

Our Framework An effective framework for verifying peephole optimizations should support *local reasoning*: a peephole optimization is a local program rewrite that preserves the machine-state transitions of a sequence of instructions. Formal compiler verification ultimately requires showing that such a local transformation preserves *global* program meaning. Therefore, we have developed a framework with a "once and for all" proof that any peephole optimization preserving local meaning also preserves global meaning.[1] Section 3 discusses the proof obligations a peephole-optimization writer must meet to use our framework. Section 4 presents a theorem that lifts these local proofs to global correctness guarantees.

Because peephole optimizations rely extensively on liveness information (whether a register value will be used before it is overwritten), our framework provides a verified liveness analysis. A peephole optimization indicates which registers need to be dead for the optimization to apply, allowing it to change the values in temporary (dead) registers.

Finally, we provide a verified program-transformation engine that takes a peephole optimization and applies it where possible in a program.

Contributions Verifying peephole optimizations in CompCert required several technical advances, which we provide in a new framework named Peek. These advances include:

- A new CompCert x86 assembly semantics that represents pointers as 32-bit integers, and a proof of equivalence with the existing, higher-level assembly semantics under reasonable and implementable assumptions about the system memory allocator.
- A liveness analysis over x86 assembly, verified for all assembly programs adhering to the standard C calling convention (see Section 4.3).
- A modular framework in which each correct concrete peephole optimization is shown to preserve global program meaning.
- A library of lemmas and tactics that allow for easy verification of additional peephole optimizations.
- A technique for parameterizing concrete peephole optimizations over register names to work for any correct instantiation of registers.

As detailed in the paper, our advances come with a few caveats that we believe could be addressed in future work without significantly modifying our overall approach. For example, we require peephole optimizations to preserve program length, inserting no-ops rather than shortening the program.

To evaluate our work, we implemented and verified 28 optimizations capturing a range of idioms as well as examples from the superoptimization literature [2]. All our code and proofs are publicly available.[2]

[1] Our current proof makes an explicit assumption that all functions in the program follow the standard calling convention (see Section 4.3 for details), which could be proven in future work without altering our framework.

[2] https://github.com/uwplse/peek

2. Overview

Background and Example The peephole optimizations we consider operate over compiler-generated assembly code, i.e., at the lowest-level language used in the compiler. These transformations can take advantage of machine-specific information and can clean up inefficiencies introduced during the translation to assembly.

As a simple-but-clever example, consider this rewrite for x86 assembly[3] from the literature [2]:

```
sub %eax, %ecx            notl %eax
mov %ecx, %eax   =====>   add %ecx, %eax
dec %eax
```

We call the code on the left the *find-pattern* and the code on the right the *replace-pattern*. This transformation corresponds to replacing the expression `x = y - x - 1` with `x = y + ~x` where `x` and `y` are twos-complement 32-bit integers initially in registers `%eax` and `%ecx` respectively and `~` is bitwise-negation. While it could be applied in any intermediate representation (that has bitwise negation), doing so misses opportunities since the pattern can arise in generated code for operations like indexing into memory structures—code not explicit in higher-level representations. Other examples of peephole optimizations may use assembly-language features (e.g., vector instructions) unavailable at higher levels.

To perform a peephole optimization, a compiler scans the assembly code for the find-pattern (allowing various permutations of registers) and replaces it with the replace-pattern. Compilers typically have many different peephole optimizations, with an interface for expressing new ones. An execution engine performs the scanning-and-replacing for any peephole optimization.

Our peephole-optimization framework for CompCert follows this typical architecture, with proofs that (1) each peephole optimization is correct and (2) the full system is correct if each peephole optimization is correct.

Local Correctness and the Role of Liveness Informally, we reason that a peephole optimization is correct if the find-pattern and the replace-pattern act identically on the machine-state: for all states, applying either instruction sequence results in the same state.[4] However, requiring precisely identical state is too strong: registers in an assembly program are often *dead*, meaning they are not read before next being written. An optimization need not preserve the values in such registers. In our example above, the transformation does not preserve the value in `%ecx`, so it is valid only in contexts where `%ecx` is dead.

[3] For the duration of the paper, we will use GAS syntax.

[4] As a technical detail, this clearly cannot hold if the program text is part of the state, which is relevant in the presence of self-modifying code, programs that observe the size of the binary or the exact value of the program counter, etc. Compilers can ignore these issues because such techniques for inspecting the binary are unavailable to well-defined source programs.

In unverified compilers, peephole optimizations can often assume certain registers are dead, using knowledge about compiler-specific code-generation details. Examples include knowing that certain registers may only be used for temporary values, or flag registers are never live across a branch. Our verified system has no such luxury, so each peephole optimization lists the set of registers that must be dead for it to apply. Furthermore, our framework includes a verified intraprocedural static liveness analysis over the assembly code to produce a sound approximation of which registers must be dead at each program point. The execution engine uses information from this liveness analysis to apply a peephole optimization only when the registers it requires to be dead are, in fact, dead.

A Lower-Level Assembly Semantics Every intermediate language in CompCert has a dynamic semantics. The compiler does not execute these semantics; it only generates assembly code to be run later. The semantics are instead used to prove that each optimization or translation to a lower-level language preserves meaning. The semantics for the highest- (C) and lowest-level language (x86) are particularly important because they are *trusted* to adequately represent the C language definition and the x86 language definition.

As explained in Section 5, there is a semantic gap between CompCert's definition of 32-bit x86 assembly and reality. CompCert's x86 semantics represents pointers as a pair of a memory block and an offset, not as the 32-bit integers used in reality. Blocks are represented by mathematical integers, and memory is assumed to be infinite.

CompCert's x86 semantics correspondingly produces undefined values for bitwise (and other) operations on pointers. With such a semantics, the example transformation above is invalid when `%eax` holds a pointer. But in reality, this restriction is unnecessary: the optimization is correct for any register contents, even if the bits happen to represent a pointer. Therefore, we wish to prove such optimizations correct without concern for the type of value in the register.

To achieve this, we defined a new lower-level x86 semantics in which pointer values are 32-bit integers. This allows more peephole optimizations to be proven correct. This additional semantics makes explicit several assumptions made by CompCert. Instead of assuming an infinite memory, memory allocation must be modeled. Memory allocation is allowed to fail since only finitely many pointer values exist. We prove that our semantics is equivalent to the existing one on any execution where memory allocation does not fail and the memory allocator obeys several sensible axioms (as enumerated in Section 5). Our new assembly semantics provide a more realistic model of x86 in CompCert's trusted computing base, and the equivalence proof preserves CompCert's existing end-to-end guarantee.

Global Correctness To maintain CompCert's compiler-correctness proof, we must show that applying any (locally) correct peephole-optimization preserves *global* pro-

gram meaning. This proof contains three orthogonal components, each discussed in detail Section 4:

- The liveness analysis is sound: any register identified as dead at a program point must actually be dead whenever execution reaches that program point.
- If a peephole optimization is locally correct, then any use of it at any place in the program would preserve program meaning.
- The execution engine performs peephole optimizations correctly: it performs a rewrite as specified and only at program points where the optimization is valid.

Each of these proofs uses the new x86 assembly semantics. A separate proof guarantees that this new semantics is equivalent to the prior infinite-memory semantics if memory allocation does not fail.

Verification and Assumptions. All our proofs are in Coq, using the existing CompCert framework. Our liveness analysis assumes (but does not verify) that all code obeys the C calling convention (see Section 4.3 for details). Our semantics for memory allocation axiomatizes assumptions about the memory allocator rather than verifying an actual memory allocator. We believe these assumptions about the calling convention and memory allocator could be discharged independently of our work.

3. Proving Peepholes

Peek eases the proof burden for formally verifying peephole optimizations by identifying a set of *local properties* that are practical to prove for an optimization yet sufficient to establish the global correctness of applying any optimization. This section details the three parts of such records: *Peephole Data*, which specifies the transformation, *Local Correctness*, which proves the transformation satisfies the local correctness properties, and *Side Conditions*, which establish auxiliary properties used in the global correctness proof.

3.1 Peephole Data

A *concrete* peephole transformation is defined by a list ℓ of x86 instructions to find, and a list r of x86 instructions to replace them with, and 3 sets of registers: U, D, and T. U lists all registers that ℓ and r read, D lists all registers that ℓ and r update (with equal values), and T lists all registers that ℓ and r may leave with different contents (i.e. scratch registers the peephole can trash). This set is necessary for peepholes for which the r pattern uses fewer registers than the ℓ pattern, such as converting 3 move instructions to an exchange. Thus, a register that contains an input value to the peephole and also later contains an output value, could be in both U and D, and any register that is written by either ℓ or r must be in D or T. The registers in U and D could be computed with static analysis, but Peek currently requires the peephole writer to specify them, which we have found easy to do in practice.

The problem with concrete transformations is that we would need an almost-identical transformation for every combination of registers. Instead, we want *parameterized* peephole transformations that can be instantiated with actual registers to produce concrete transformations. A parameterized peephole transformation is a function from register names to a concrete peephole transformation. Verifying a parameterized peephole transformation requires proving that, for any instantiation of register names, the resulting concrete peephole transformation will be correct as described the following subsections. As described later in Section 4, rewriting with a parameterized peephole transformation requires computing a substitution to instantiate the register name parameters in the rewrite.

3.2 Local Correctness

In Peek, a concrete peephole transformation is correct if ℓ and r always terminate and make equal updates to all live locations, assuming they begin in states that agree on all live locations. Control is implicitly preserved by this definition, because the program counter is always live. *Local correctness* establishes that ℓ and r make equal updates to all registers in D and equal updates to all memory locations, assuming that they begin in states that agree on all registers in U. Let P be the set of registers that do not appear in U, D, or T; these are registers that neither ℓ nor r read or write and are thus preserved by the peephole. The Peek execution engine "ties the knot" by computing the sets of live locations L_{entry}, L_{exit} at the entry and exit of rewritten locations respectively and checking that both $U \subseteq L_{entry}$ and $L_{exit} \subseteq D \cup P$. Note that because local correctness requires equal memory updates from ℓ and r, we do not compute liveness for memory locations.

Let $\sigma \overset{c}{\rightsquigarrow} \sigma'$ hold whenever a state σ executes through code fragment c (within a larger program) to yield state σ' and let $\sigma_1 \overset{L}{\sim} \sigma_2$ hold when σ_1 and σ_2 agree (as defined in Section 4) on all locations in L. Then local correctness can be proved by showing (1) *local simulation* that holds for any larger program context ℓ and r may appear in:

$$\forall\ \sigma_\ell\ \sigma'_\ell\ \sigma_r,\ \sigma_\ell \overset{U}{\sim} \sigma_r \wedge \sigma_\ell \overset{\ell}{\rightsquigarrow} \sigma'_\ell \implies \exists\ \sigma'_r,\ \sigma'_\ell \overset{D}{\sim} \sigma'_r \wedge \sigma_r \overset{r}{\rightsquigarrow} \sigma'_r$$

and (2) *ℓ-normalization* by providing a measure m from machine states to natural numbers satisfying:

$$\forall\ \sigma_\ell\ \sigma'_\ell,\ \sigma_\ell \overset{\ell}{\rightsquigarrow} \sigma'_\ell \implies m(\sigma'_\ell) < m(\sigma_\ell)$$

For intuition, *ℓ-normalization* is simply a proof that any execution which enters the ℓ pattern will not diverge within it.

Consider proving local simulation for the example in Section 2: σ_l and σ_r have the same initial values in `%eax` and

%ecx, call them a_0 and c_0. For σ_l, the final value in %eax is $c_0 - a_0 - 1$. For σ_r, the final value in %eax is $c_0 + \tilde{}a_0$. As D contains only %eax, local simulation amounts to proving $c_0 - a_0 - 1 = c_0 + \tilde{}a_0$, which holds for 32-bit ints.

Because we require only a local simulation, peephole optimizations cannot rely on any properties of the rest of the program, including position in the program, invariants about values in particular registers, etc. This is the essence of what makes an optimization a *peephole* optimization. This interface provides modularity by allowing the rewrite to be carried out in many contexts. Section 4 details our proof showing that for any context, local simulation relations are sufficient to establish global optimization correctness.

Regarding ℓ-normalization, we currently support the common case where ℓ contains only forward jumps (or no jumps). Peek provides a default measure (difference between PC and end of the rewrite), and a proof that this measure implies normalization. Supporting rewrites that contain backwards jumps would require manipulating a measure construction provided by the optimization implementer.

Note that there is no explicit proof that the r pattern normalizes. However, the local simulation relation already accomplishes this, since if the ℓ pattern normalizes, then there is some finite series of steps from the beginning to the end of the ℓ pattern. The local simulation relation relates those steps to a finite series of steps from the beginning to the end of the r pattern. Thus the r pattern also normalizes.

3.3 Side Conditions

The global proof of correctness (Section 4) assumes rewrites satisfy the following side conditions: the length of ℓ is the length of r, [5] and that the final instruction in ℓ is not a label.[6] Furthermore, Peek requires that ℓ and r do not contain any call or return instructions, which implies that they do not modify the global program trace by making a system call. In practice, this condition could be relaxed but no peephole transformation we found violated the constraint and this condition eases the global correctness proof. All the above conditions are decidable, and Peek includes decision procedures or tactic support for automating their proofs.

3.4 Local Peephole Verification

As described above, the framework requires both a termination measure and a local simulation. In practice, simply using the number of instructions remaining in a peephole suffices for the measure of all peepholes we have verified. Proofs of local simulation are, on the other hand, very specific to each peephole, but with some common elements. They naturally decompose into two parts: (1) proving the right side can step (given the steps on the left), and (2) proving the resulting values computed by the right and left are equivalent.

To ease proving (1), we developed lemmas and custom Ltac tactic support. Without this support, naively constructing a single step on the right side can take between 60 and 100 lines of Ltac and exhausted available memory on some optimizations as short as 4 instructions. By developing tactics specifically related to correlating the resulting left and right program states, a single tactic often suffices for each program step, typically taking under a second and imposing reasonable memory overhead. Furthermore, this tactic support can typically automatically prove that the right-hand-side can step, allowing the optimization writer to focus on (2) and show that the optimization is meaning preserving.

4. Verifying Peek

Section 3 discussed the local properties sufficient for a correct peephole transformation. This section connects these local properties to global program execution. As noted in previous sections, liveness plays a key role in verifying Peek. The high level approach to the main correctness result is implementing a liveness analysis that computes the set of live registers at every program point, proving that any transformation that preserves the values of these registers also preserves global program meaning, and then proving that any application of the execution engine to apply a locally correct peephole will preserve live locations and therefore global program meaning. We detail each of these components and their proofs below.

4.1 Liveness

A peephole's local correctness properties imply global correctness only when the liveness information at a rewritten program location guarantees that all the inputs (registers containing values used in the peephole) are live and that all live locations at the output that are updated by the find pattern are refined by the replace pattern as detailed below.

Our liveness analysis computes a function live mapping program locations to a set containing all live registers at that location. We prove that a program's behavior depends only on the values stored in registers marked as live. We say y *refines* x if either x is undefined or $x = y$. Further, if for each register r in L, the value contained in register r in σ_2 refines the value contained in r in σ_1, we write $\sigma_1 \overset{L}{\sim} \sigma_2$. If $\sigma_1 \overset{L}{\sim} \sigma_2$, then for any correct liveness set L, program executions from these states will generate equivalent traces of externally visible events. Let $\sigma(r)$ denote the value of register r in state σ. We show that any transformation preserving liveness also preserves program behavior by proving this simulation

[5] This restriction could be relaxed with a pre and post pass to insert and/or remove NOP instructions. We have not verified NOP insertion or deletion, but we have defined the CompCert pretty printer to print NOP instructions as the empty string, effectively implementing NOP removal.

[6] This last detail arises from CompCert's choice to make labels themselves instructions and to model a jump to a label as transferring control to immediately after the label.

relation:

$$\forall\ \sigma_1\ \sigma_1'\ \sigma_2,\quad (\sigma_1 \overset{\texttt{live}(\sigma_1(\texttt{PC}))}{\sim} \sigma_2 \quad\wedge\quad \sigma_1 \rightsquigarrow \sigma_1') \implies$$
$$\exists\ \sigma_2',\quad \sigma_1' \overset{\texttt{live}(\sigma_1'(\texttt{PC}))}{\sim} \sigma_2' \quad\wedge\quad \sigma_2 \rightsquigarrow \sigma_2'$$

4.2 Liveness Implementation

Peek's liveness analysis is an iterative dataflow analysis. The set of live registers is initialized to $\emptyset$ at each program point, and an update function is iteratively applied using a standard worklist algorithm until the liveness information reaches a fixed point with respect to the update function. The liveness at return instructions is fixed to callee-saved registers and the register containing the return result. Likewise, the liveness at function call sites is fixed to caller-saved registers.

The proof that the calculated liveness information meets the above specification naturally decomposes into two parts. First, we prove that if iteration of the update function halts, then the result is a fixed point of the update function. Specifically, the calculated liveness information is invariant to future applications of our update function. Second, we prove that liveness information that is at a fixed point with respect to the update function is a simulation relation (as above). To prove this, we show that our static approximation to control flow is sound, and that our static approximation to which registers each instruction uses and defines is sound. These two facts compose nicely into a simulation relation.

4.3 Calling Convention

The correctness of the liveness analysis depends on five facts that comprise the x86 calling convention that CompCert uses. They are true for CompCert-generated code, but we have not verified them. These are necessary for proving the intraprocedural analysis is interprocedurally correct, as the given CompCert semantics for call and return instructions are defined quite liberally, and do not provide many guarantees.

1. Any step that executes a call instruction steps to the beginning of a block (i.e., the beginning of a function).
2. Any step that returns from a function steps to an instruction right after some call.
3. All non-callee-save registers are dead when executing a call instruction. (CompCert passes all function parameters on the stack.)
4. All registers except the callee-save registers and the return-value register are dead when executing a return.
5. When returning, the location of the return value assumed by the returning function matches the location of the return value assumed by the caller.

4.4 Global Peephole Optimization Correctness

The liveness analysis and local peephole properties come together to form a correctness proof of global program transformation. The proof of global correctness takes a local peephole simulation relation and forms a global program simulation relation with input from the liveness analysis. We use CompCert's approach for proving program transformations correct. In CompCert, a transformation is correct if there is a bisimulation between the original and transformed programs. In practice in CompCert, we prove only a forward simulation for each transformation and use the fact that each intermediate language's semantics is deterministic to construct a backward simulation. Therefore, given two programs p_1 and p_2, we need to show that all possible behaviors of p_1 are possible behaviors of p_2.

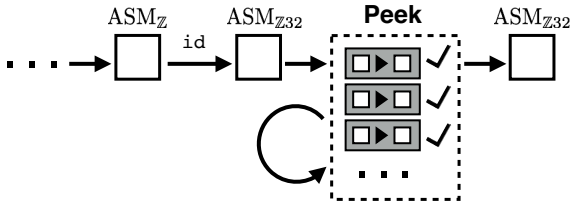

Figure 1. Peek Backend CompCert produces code in ASM$_{\mathbb{Z}}$. We prove a forward simulation to establish that a program will have equivalent behavior under ASM$_{\mathbb{Z}32}$. Peek repeatedly applies peephole optimizations from a set of verified rewrites to the code.

A forward simulation shows that if state σ_l in the original program matches some state σ_r in the transformed program, and σ_l can take a step to σ_l', then there is some state σ_r' that σ_r can step to, and the resulting states σ_l' and σ_r' match. This relation can be transitively composed to prove that any executions of arbitrary length agree between the original and transformed program.

For proving peephole rewrites correct globally, we partition program states as follows. A state σ is *outside* a rewrite if the program counter points to an instruction that was not rewritten by the transformation. A state σ is *at entry* if the program counter points to an instruction that is the first instruction that was rewritten by the transformation. A state σ is *inside* if its program counter puts it at an instruction that was rewritten, but is not the first instruction in the rewrite, and there is a series of steps from some other state σ_0 to σ, where σ_0 is *at entry*, and those steps occur within the rewritten region. (See Figure 2.)

It remains to define this matching relation between σ_l and σ_r to show that any locally correct peephole can be applied to produce a new program that (forward) simulates the original program. σ_l and σ_r are in the relation if any of the following hold:

- *match out*: σ_l and σ_r are both at the same program location, that program location is *outside* as defined above, σ_l and σ_r contain the same values in all live registers, and σ_l and σ_r have identical memory.

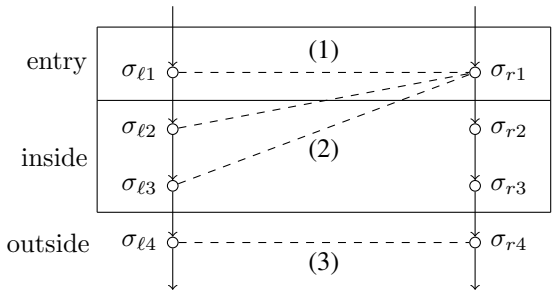

Figure 2. Match States Cases: At the beginning of the rewritten region (1), states match when at the same code location. As σ_ℓ steps through the left side of the rewritten region, it repeatedly matches σ_{r1} (2). Once σ_ℓ exits the rewritten region, it again matches at identical code locations (3).

- *match entry*: σ_l and σ_r are both at the same program location, that program location is *at entry* as defined above, σ_l and σ_r contain the same values in all live registers, and identical memory.
- *match in*: σ_l is *inside* as defined above, σ_0 and σ_r contain the same values in live registers, and the same memory.

The entry and outside cases, while similar, are kept distinct to allow for the **Step Cases** lemma below. To apply a peephole transformation, we also need to prove that control flow enters the rewritten region only at the beginning of the rewrite and exits only at the end. Without this property, very few classical peephole transformations are valid. Proving this fact is surprisingly subtle in x86, as some instructions have semantics that simply store the value of a register into the program counter and thus could jump to arbitrary locations.

We prove this control-flow property via two lemmas: *single entry* states that any execution step that would enter the rewritten region always steps to the first instruction, and *single exit* states that any execution step that would exit the rewritten region steps to the instruction immediately following the rewritten region. Using these lemmas, we characterize the scenarios a program can be in after taking a step:

Step Cases Lemma: If $\sigma \rightsquigarrow \sigma'$, then (1) if σ is *outside*, then σ' is *outside* or *at entry*, and (2) if σ is *at entry* or *inside*, then σ' is *inside* or *outside*.

This lemma then allows the simulation proof to proceed by handling the corresponding cases. The following theorem is the proof of global correctness.

Theorem: If σ_l matches σ_r, and $\sigma_l \rightsquigarrow \sigma'_l$, then either there is some σ'_r such that $\sigma_r \rightsquigarrow \sigma'_r$ and σ'_l matches σ'_r, or σ'_l matches σ_r and a measure decreases from σ_l to σ'_l.
Proof Sketch:

- If σ_l is *outside*, and σ'_l is *outside* or *at entry*, the proof follows from the correctness of liveness.
- If σ_l is *at entry* or *inside*, and σ'_l is *inside*, the proof shows that σ'_l matches σ_r and the measure decreases. The proof of the measure decreasing comes from local properties in the verified rewrite rule.
- If σ_l is *at entry* or *inside*, and σ'_l is *outside*, we first appeal to the *single exit* property to show that σ'_l is precisely at the end of the rewritten region. Next, we unfold the definition of *inside*, collect the series of steps from our original σ_0 to σ'_l, and use that series of steps to leverage the correctness of the rewrite.

4.5 Parameterized Rewrites

Recall parameterized rewrites work for any set of appropriate registers whereas concrete rewrites specify exact assembly registers. We use higher-order functions to get the expressiveness of parameterized rewrites with the verification burden of concrete rewrites as follows: The optimization writer actually writes a function that given an instruction sequence (by the optimization-execution engine) determines if registers can be assigned to produce a verified concrete transformation that matches the instruction sequence. If so, the particular correct concrete transformation corresponding to the code found in the program is produced. The proofs of correctness of these transformations are done once and for all, with abstract registers and constraints over them. Once a concrete rewrite is needed, the particular proof is simply instantiated with the given concrete registers and appropriate proofs of register constraints. Note the function that attempts to produce the concrete transformation need not be verified itself — if it wrongly misses an opportunity to produce a concrete transformation no unsoundness results, and the type system guarantees that the optimization writer cannot write a matcher that ever successfully produces a concrete transformation that is not correct. In practice, writing a parameterized transformation (which is what peephole transformations in conventional compilers actually are) is hardly more difficult than writing a concrete transformation: one simply pattern-matches on an instruction sequence and checks register-name constraints. We include tactic support to further minimize the effort.

5. Semantics Over Bits

In CompCert, many correct peephole optimizations are not provable because arithmetic operations are undefined (or partially defined) over pointer values. Peephole optimizations must be correct regardless of the type of value a register contains at runtime. To support peepholes, we add a new CompCert assembly semantics where pointers are represented as concrete 32-bit integers so more operations will be defined over pointers.

This section motivates the need for this change, defines the new language, axiomatizes a correct memory allocator that can fail due to memory exhaustion, and describes the proof that our semantics is equivalent to the existing se-

mantics on any execution where memory allocation always succeeds. We refer to the "existing" semantics with respect to infinite memory as $\text{ASM}_{\mathbb{Z}}$ and our "new" semantics as $\text{ASM}_{\mathbb{Z}32}$ since the change is to give pointers a bit vector representation.

5.1 Motivation

In CompCert C, the (non-aggregate) values are floats of various sizes, integers of various sizes, pointers, or the undefined value. The dynamic semantics of every intermediate language in CompCert is defined to compute over these same values, with each value containing a (runtime) type tag. These type tags are included even in $\text{ASM}_{\mathbb{Z}}$ (and retained in $\text{ASM}_{\mathbb{Z}32}$) even though an actual x86 implementation does not have them: on actual hardware, there are only bit vectors without the corresponding type tags. Thus $\text{ASM}_{\mathbb{Z}32}$ guarantees that, at runtime, no values with tag `Vptr` are present in registers or memory, all memory addresses are represented using `Vint`.

This semantic gap then propagates to the semantics of instructions. With CompCert's type tags, the instruction definition often branches on the tag of an argument's value with some branches producing the special "undefined" value `Vundef`. An example of such a semantics is for the integer-subtraction instruction:

```
Definition sub (v1 v2: val): val :=
 match v1, v2 with
  | Vint n1, Vint n2 =>
      Vint(Int.sub n1 n2)
  | Vptr b1 ofs1, Vint n2 =>
      Vptr b1 (Int.sub ofs1 n2)
  | Vptr b1 ofs1, Vptr b2 ofs2 =>
      if eq_block b1 b2
      then Vint(Int.sub ofs1 ofs2)
      else Vundef
  | _, _ => Vundef end.
```

In actual x86, this subtraction instruction always performs subtraction on bit vectors. In the definition above, several cases instead produce `Vundef`, such as if an operand is a float or if the two operands are pointers into different blocks. The $\text{ASM}_{\mathbb{Z}}$ semantics is sufficient because (1) legal C source programs will not depend on undefined-values[7] and (2) when undefined-values are not produced, the semantics aligns with actual x86 just with the addition of type tags and with a higher-level representation of pointers using a block and an offset (see Section 5.2).

Now consider the peephole optimization first shown in Section 2 that uses bitwise negation to replace subtraction. The semantics for taking the bitwise-negation of a pointer is `Vundef` since there is no reasonable way to bit-negate a pair of a block (represented as a $\mathbb{Z}$) and an offset (represented as a $\mathbb{Z}_{32}$), nor is the operation defined on C pointers. But now under this semantics, the peephole optimization is *not* meaning-preserving due to program states where an operand may be a pointer. A peephole optimization must be correct for all (types of) operands since by this stage in the compiler, static type information has been erased. In this case, if type information had not been erased, the information available in CompCert still cannot distinguish between pointers and integers. Hence the proof of correctness for a common integer optimization, for example, must include a case for registers that contain floating point values, and registers with undefined values, and so on.

After pursuing several design alternatives (not discussed due to space constraints), we concluded that the best small-but-sufficient design change was to introduce a lower-level semantics that still has type tags but also gives pointers a bit vector representation. Two computations over pointer values are equivalent as long as they produce the same bit vector for every input. As described in the remainder of this section, we axiomatize how a memory allocator must behave in terms of the bit vectors it returns as pointers in order for $\text{ASM}_{\mathbb{Z}32}$ to be equivalent to $\text{ASM}_{\mathbb{Z}}$. We need to take particular care not to introduce a false axiom because there are finitely many 32-bit bit vectors whereas $\text{ASM}_{\mathbb{Z}}$'s semantics assumes an infinite memory, so any claim of a bijection between the two representations of pointers is logically inconsistent. Moreover, our finite range of pointer values necessitates allowing for memory exhaustion (i.e., failed allocation), a possibility not previously considered in CompCert.

5.2 Converting Between Memory Addresses and Pointer Values

In $\text{ASM}_{\mathbb{Z}}$ and $\text{ASM}_{\mathbb{Z}32}$, a *memory address* is a $\texttt{block} \times \texttt{offset}$ pair, where a `block` is a positive integer, unbounded in size, and an offset is a 32-bit unsigned integer. In $\text{ASM}_{\mathbb{Z}}$, a *pointer value* is a memory address, but in $\text{ASM}_{\mathbb{Z}32}$ a pointer value is a 32-bit integer. The two semantics use the same memory model with an infinite memory of blocks, thus simplifying the translation from $\text{ASM}_{\mathbb{Z}}$ to $\text{ASM}_{\mathbb{Z}32}$.

Performing memory operations in $\text{ASM}_{\mathbb{Z}32}$ requires conversion between 32-bit integers and memory addresses, i.e. a memory allocator. We use `pinj` ("pointer inject") to map from a memory address to a pointer value and `psur` ("pointer surject") to map from a pointer value to a memory address. As discussed below, these functions are defined as opaque axioms subject to constraints. They have these signatures:

$$\texttt{pinj} : \texttt{AS} \to \texttt{block} \to \texttt{int}_{32} \to \texttt{option int}_{32}$$

$$\texttt{psur} : \texttt{AS} \to \texttt{int}_{32} \to \texttt{option (block} * \texttt{int}_{32}\texttt{)}$$

The option in `pinj`'s return type models memory exhaustion and is crucial to avoid axiomatizing false by claiming that an infinite set can map injectively to a finite set. `psur` has a return type of option as well, which lets us axiomatize that any successful `psur` implies a successful `pinj`. To model these mappings changing over the execution of the program

[7] The subtraction example in particular matches C's semantics: Subtraction of two pointers returns a defined value only if the two pointers are into the same block of memory.

(as pointer locations are reused via malloc and free), we add an additional argument to pinj and psur, representing the current allocator state (AS). This allocator state is kept opaque, and is manipulated by malloc and free opaque functions to produce modified allocator state.

We axiomatize the propagation of the opaque allocator state. A single object of this type represents an initial (empty) allocator state, init : AS. We also assume the initialized allocator state corresponds to the initial memory when main begins to execute. We axiomatize functions representing allocate and free actions which transform the allocator state. alloc : AS $\rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow$ block $\rightarrow$ AS and free : AS $\rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow$ block $\rightarrow$ AS. In CompCert, allocate and free operate on ranges of offsets within blocks. In order to support this, alloc and free each take two $\mathbb{Z}$ arguments, which represent the range within the block. Likewise, we axiomatize as_ec and as_ec', which record external calls in the allocator state (corresponding to CompCert's external_call and external_call' respectively).

Further, we define what it means for a memory and an allocator state to *match*. Specifically, the empty memory and init match. A new memory produced from an allocate action matches a new AS produced from an alloc action (with the same arguments and results), provided the previous memory and AS matched. A new memory produced from a free action matches a new AS produced from a free action (with the same arguments), provided the previous memory and AS matched. A new memory produced from some action that does not allocate or free (e.g., store) matches any AS the previous memory matched. Again, matching allocator state and memory is similarly defined over external calls. Finally, we say that a particular AS *extends* another AS if one was produced in some number of steps from the other.

To reason about these opaque functions, which map memory addresses to $\text{ASM}_{\mathbb{Z}32}$ pointer values, we use a set of axioms about pinj and psur Figure 3.

The semantics for memory-load and -store instructions uses pinj and psur to manipulate registers holding bit vectors that describe memory addresses. Additionally, in $\text{ASM}_{\mathbb{Z}}$, a new memory allocation generates a pointer to a new block of memory in a simple way: a global counter of blocks is updated, and the new fresh block is used to generate pointers into the memory. In $\text{ASM}_{\mathbb{Z}32}$, the same approach is used, but the result pointers are injected to bits using pinj.

In $\text{ASM}_{\mathbb{Z}}$, arithmetic operations produce fewer undefined results for 32-bit integers than for pointers. That is, if an operation does not produce Vundef for pointer operands, then it does not produce Vundef for int operands. Therefore, $\text{ASM}_{\mathbb{Z}32}$ is strictly more defined than $\text{ASM}_{\mathbb{Z}}$, as everywhere $\text{ASM}_{\mathbb{Z}}$ will not produce Vundef, $\text{ASM}_{\mathbb{Z}32}$ will not produce Vundef. However, $\text{ASM}_{\mathbb{Z}32}$ will not produce Vundef in additional cases such as bitwise-negation of a pointer value, which is the entire purpose of $\text{ASM}_{\mathbb{Z}32}$.

1. pinj always remembers a mapping:
   $\forall\, as, \forall\, as', \forall\, b, \forall\, o, \forall\, bits,$
   pinj $as\ b\ o =$ Some $bits \rightarrow$ extends $as\ as' \rightarrow$
   pinj $as'\ b\ o =$ Some $bits$.
2. pinj allows for pointer arithmetic:
   $\forall\, as, \forall\, b, \forall\, o, \forall\, bits,$ pinj $as\ b\ o =$ Some $bits \rightarrow$
   $\forall x,$ pinj $as\ b\ (o + x) =$ Some $(bits + x)$.
3. pinj is injective within the same block:
   $\forall\, as, \forall\, b, \forall\, o1, \forall\, o2, \forall\, bits,$
   pinj $as\ b\ o1 =$ Some $bits \rightarrow$ pinj $as\ b\ o2 =$ Some $bits \rightarrow$
   $o1 = o2$.
4. Null is always invalid:
   $\forall\, as, \forall\, m,$ match $as\ m \rightarrow$
   $\forall\, b, \forall\, o,$ pinj $as\ b\ o =$ Some NULL $\rightarrow$
   valid $m\ b\ o$ = false $\wedge$ valid $m\ b\ (o - 1)$ = false.
5. psur is equivalent to pinj and weakly valid:
   $\forall\, as, \forall\, m,$ match $as\ m \rightarrow \forall\, b, \forall\, o, \forall\, bits,$
   psur $as\ bits =$ Some (b,o) $\longleftrightarrow$
   (pinj $as\ b\ o =$ Some $bits \wedge$ weakvalid $m\ b\ o$ = true).
6. pinj supports pointer comparison:
   $\forall\, as, \forall\, b, \forall\, o1, \forall\, o2, \forall\, bits1, \forall\, bits2,$
   pinj $as\ b\ o1 =$ Some $bits1 \rightarrow$ weakvalid $m\ b\ o1$ = true $\rightarrow$
   pinj $as\ b\ o2 =$ Some $bits2 \rightarrow$ weakvalid $m\ b\ o2$ = true $\rightarrow$
   $(o1 < o2 \longleftrightarrow bits1 < bits2)$.

Figure 3. All of the axioms constraining pinj and psur, except those related to external calls, which are used to verify the forward simulation from $\text{ASM}_{\mathbb{Z}}$ to $\text{ASM}_{\mathbb{Z}32}$.

5.3 Axiomatizing Memory Allocation

Conceptually, the system memory allocator (e.g., malloc, alloca, or function stack frames) is an instantiation of pinj and psur: Allocating a new memory block amounts to assigning it a pointer-value (its position in the virtual address space). We do not define our semantics in terms of an actual instantiation of pinj and psur, but instead we define our required specification of the allocator. (And since this is only the semantic definition, we do not actually need one. Like in CompCert prior to our work, the compiled binary is linked against the ordinary standard (unverified) library for C.)

Additionally, $\text{ASM}_{\mathbb{Z}32}$ includes the axiom that any external call behavior is duplicated when all pointer values in the registers and memory have been converted via pinj to bits, and that the allocator state produced by these two calls is equal.

These axioms comprise the specification of a memory allocator, as shown in Figure 3. Any allocator must satisfy all of the properties above, and any allocator satisfying those properties is a valid system allocator for running assembly programs. For example, since pointer arithmetic exists at the assembly level, the allocator must preserve the behavior of that arithmetic (2); the allocator must not use the NULL address for anything, as programs rely on that address being invalid (4); and pinj and psur must be inverses of each other,

as it is essential for the semantics to correctly pack/unpack pointer values (5).

Memory deallocation (i.e, `free`) deserves brief discussion. Because CompCert's correctness guarantee is only for legal C programs and programs executing dangling-pointer dereferences are illegal, their behavior is not directly relevant. In $\text{ASM}_{\mathbb{Z}}$ and $\text{ASM}_{\mathbb{Z}32}$, `free` marks a block of memory as unusable and subsequent access produces `Vundef`. More interesting, and unconsidered in $\text{ASM}_{\mathbb{Z}}$ due to infinite memory, is needing to *reuse* pointer values. Our explicit handling of allocator state has allowed us to build, to our knowledge, the first formally verified system with a specification of a memory allocator which allows for memory reuse.

5.4 Equivalence of $\text{ASM}_{\mathbb{Z}}$ and $\text{ASM}_{\mathbb{Z}32}$

$\text{ASM}_{\mathbb{Z}32}$ is a new semantics over the same syntax for which $\text{ASM}_{\mathbb{Z}}$ is defined. To maintain CompCert's correctness guarantee, a bisimulation proof between $\text{ASM}_{\mathbb{Z}}$ and $\text{ASM}_{\mathbb{Z}32}$ is required. We have constructed this proof, *assuming that pointers produced in the program by $\text{ASM}_{\mathbb{Z}}$ are injectable to bits:* i.e., do not return `None` when passed to `pinj`. If `None` is produced, this corresponds to a memory-allocation failure, which is allowed in C but not modeled by $\text{ASM}_{\mathbb{Z}}$ or any of CompCert's higher-level languages. More formally, we assume that any pointer to the first byte of all allocated blocks in any program state reachable (in some finite number of steps) from the initial state does not return `None` when passed to `pinj`.

We only guarantee forward simulation in cases where memory allocation doesn't fail: a guarantee morally equivalent to the existing CompCert guarantee. Our approach to injecting pointers does not, in essence, add axioms to CompCert so much as it makes explicit certain assumptions which existed implicitly in the semantic gap between CompCert's x86 backend semantics, and the actual behavior of an x86 chip.

In order to argue forward simulation from $\text{ASM}_{\mathbb{Z}}$ to $\text{ASM}_{\mathbb{Z}32}$, the definition of matching register sets is straightforward: If a register contains a value other than a pointer or undefined in $\text{ASM}_{\mathbb{Z}}$, it contains that same value in $\text{ASM}_{\mathbb{Z}32}$. If a register contained a pointer in $\text{ASM}_{\mathbb{Z}}$, it contains the corresponding integer in $\text{ASM}_{\mathbb{Z}32}$. If a register contained the undefined value in $\text{ASM}_{\mathbb{Z}}$, it could contain any value (other than a pointer) in $\text{ASM}_{\mathbb{Z}32}$.

6. Evaluation

Peek is implemented and verified as an extension of CompCert version 2.4. We discuss our implementation and evaluation of it in these terms:

- The complexity of Peek's implementation, proof, and trusted computing base
- The complexity of proving particular peephole optimizations correct, including the variety of optimizations we have proven such as several challenging ones from the recent superoptimization literature
- Compiling CompCert benchmark programs with our peephole optimizations: how many programs have peephole optimizations occur and a preliminary investigation of performance impact

	Spec	Proof	Total
Peek Total	**6,000**	**10,000**	**16,000**
Liveness	1,300	2,300	3,600
Peephole Exec	3,300	6,600	9,900
Libraries	1,100	900	2,000
Parameterization	40	163	203
$\text{ASM}_{\mathbb{Z}32}$	**3,300**	**5,500**	**8,800**
Peephole Lib	**2,000**	**3,100**	**5,100**
Total	**11,300**	**18,600**	**29,900**

Figure 4. Approx. lines of code according to `coqwc` tool.

6.1 Implementing Peek

The peephole framework that we added to CompCert comprises approximately 30,000 lines of Coq code and proof lines. Its main components are the lower level x86 semantics, liveness analysis, and the peephole execution engine and their respective proofs of correctness. We also provide a library to make proving peephole optimizations correct easier. The implementation sizes of these components are shown in Figure 4.

The trusted computing base (TCB) for Peek is similar to the TCB for CompCert. Just like CompCert, we trust the Coq proof checker, the Ocaml compiler and runtime, and that the C front-end models the semantics of C. Unlike CompCert, we trust the $\text{ASM}_{\mathbb{Z}32}$ backend instead of the $\text{ASM}_{\mathbb{Z}}$ backend to model x86 execution faithfully, but as explained earlier, we proved $\text{ASM}_{\mathbb{Z}32}$ and $\text{ASM}_{\mathbb{Z}}$ equivalent. Furthermore, $\text{ASM}_{\mathbb{Z}32}$ is closer to real program execution than $\text{ASM}_{\mathbb{Z}}$. We further trust that certain axioms are true, in particular that all generated code obeys the standard calling convention. Note that adding a peephole optimization incurs no increase in our trusted computing base.

6.2 Peephole Success

CompCert ships with a collection of small benchmarks, some of which are implementations of various hash functions in C. Some of these hash functions frequently left-rotate 64 bit values. For some of the left-rotate operations, CompCert currently generates 3 instructions including a left shift, a right shift, and an or. However, a single extended shift instruction is bitwise equivalent to these three. We implemented and verified two peepholes of this style, but with different orders of shifts. With these peephole transformations enabled, the `siphash24` benchmark in the CompCert

benchmark suite sped up by 4%, and the `sha3` benchmark saw a speedup of almost 1%.

6.3 Peephole Variety

We proved a total of 28 peephole optimizations correct, using a range of different assembly computations. Some perform arithmetic operations (e.g., shift instead of multiply), others remove unnecessary jumps, eliminate redundant loads, or improve instruction-level parallelism. The appendix contains all the peephole transformations we have verified. Our purpose is to exercise a variety of assembly instructions to show that our framework is expressive enough to reason about semantic equivalence for useful transformations. Moreover, proof burden, while clearly non-trivial, is minor compared to proving the framework correct. The average proof size for a peephole optimization was 72 lines. Our approach for parameterization over registers does not increase this proof size.

Six of our verified peephole optimizations are all six of the assembly transformations presented as discovered by the stochastic superoptimizer in Bansal et al.'s recent work [2].[8] We took these rewrites as "challenge problems" that perform surprising transformations where equivalence is not obvious. Note the contribution of Bansal et al. is a system for automatically discovering them; we verified their correctness.

Qualitatively, the limiting factor in implementing more peephole optimizations, particularly those that would improve performance, is that the current $\text{ASM}_{\mathbb{Z}}$ and $\text{ASM}_{\mathbb{Z}32}$ semantics covers only the subset of x86 behavior needed by the existing code generator. It is conservative in two respects: (1) not including unused instructions or addressing modes and (2) often leaving values in flag registers undefined even when the instruction behavior for x86 specifies them. This conservatism is a common and wise engineering decision in formal verification — specify only the subset the system needs — but it directly limits the expressiveness of peephole optimizations we can verify. To date, we have been loathe to make additions to $\text{ASM}_{\mathbb{Z}}$ (and correspondingly $\text{ASM}_{\mathbb{Z}32}$), but future work can relax this approach with appropriate (unverifiable) care that our additions model x86 properly.

6.4 Benchmark Results

While our focus has been on verification and expressiveness, we also compared the output of CompCert with and without our optimizations on a standard set of small benchmarks shipped with CompCert. On 15 of the 23 benchmarks, none of our optimizations apply, so identical assembly is generated. For the remaining, see Figure 5 for the number of optimizations that occur (static count), and the percent speedup.

Performance improvements so far are less encouraging. We see improvements of 4.0% and 0.7% on benchmarks `siphash24` and `sha3` respectively. For other benchmarks, run-time does not change to a statistically significant extent.

Benchmark	# of Rewrites	Speedup
`binarytrees`	1	-
`chomp`	6	-
`fannkuch`	3	-
`fftw`	1	-
`knucleotide`	10	-
`sha1`	4	-
`sha3`	23	0.7%
`siphash24`	33	4.0%

Figure 5. Performance results of Peek.

These benchmarks were run on an intel i7-4790K at 4.00GHz with 16 gigabytes of RAM, running Ubuntu 15.04.

As discussed above, the limited expressiveness of $\text{ASM}_{\mathbb{Z}}$ holds back additional improvements. We have identified peephole optimizations that would speed up additional benchmarks, but verifying them would require extending $\text{ASM}_{\mathbb{Z}}$ to include new instructions and addressing modes.

6.5 Verified Benchmarks

One of the big advantages of low marginal cost verified optimization, is that it can be applied to speed up already verified code while retaining the guarantees already provided by this verified code. We took the C code that the VST [1] project verified was a correct implementation of SHA-256, and used it as an additional benchmark. We have developed verified peepholes which fire on this benchmark, and managed a 3.9% speedup (arithmetic average over 9 trials).

7. Related Work

Peek builds on previous research in peephole optimization, extensible compilers, and formal compiler verification.

Peephole Optimizers. Peephole optimizations are well-studied in the compiler literature [15], particularly in the context of superoptimization [2, 4, 14, 19, 22, 23]. However, to our knowledge, Peek is the first peephole optimization framework within a fully formally verified compiler. In particular, Peek shows how to extend CompCert's memory model to support verifying common peephole optimizations and provides a framework to formally prove peepholes with reasonable proof overhead.

Extensible Compilers. Frameworks like Gospel [28], Broadway [6], Cobalt [10], Rhodium [11], and PEC [8] all enable the programmer to express optimizations in a domain-specific language (DSL). Allowing programmers to develop optimizations in a DSL eases the effort to write an optimization and makes it easier for the framework to analyze and run the optimization. Of these frameworks, Cobalt, Rhodium, and PEC all exploit the restricted language of the DSL to au-

[8] In one case, we changed the optimizations to use a `test` instruction rather than subtraction because CompCert's semantics for subtraction is conservative with respect to how flags are set.

tomatically prove the correctness of optimizations. In these systems correctness is checked fully automatically using a solver like Z3, but both the reduction from the optimization correctness problem to Z3 queries and the execution engine are trusted to be correct without proof.

Verified Compilers. There is a long history on compiler verification, from early projects like Piton verified in ACL2 [16] to more recent work on project's like Chlipala's Lambda Tamer [5], Leroy's CompCert [12], and CompCertTSO [27]. These project all develop techniques to provide machine-checkable proofs of the compiler's correctness. Formalizing assembly languages represents some of the most relevant work from this area. Sewell et. al.'s work on formalizing x86 semantics [18] and Morrisett et al.'s x86 semantics for the NaCl SFI checker [17] serve as examples of realistic x86 formalizations. However, none of these systems provide support for extensible optimization passes; in particular, they do not support proving and adding peephole optimizations in the context of a realistic C compiler like CompCert.

The XCert tool was designed for automatically formally verifying optimizations checked by an SMT-based optimization proof tool. Unlike Peek, XCert worked in the compiler middle-end (RTL) which greatly eased implementation, reasoning, and verification of XCert. By operating at the RTL level, XCert can simply "link in" new code to avoid invalidating dataflow facts about other locations in the program. Peek has no such luxury at the assembly level where code is represented as a linear sequence of instructions in memory. More importantly, XCert is not fully formally verified. As a solver-aided tool, XCert has a much larger TCB than Peek because XCert assumes the correctness of an SMT solver like Z3. Thus XCert's TCB is bigger because it includes an SMT solver and XCert cannot verify assembly level transformations. Peek's primary goal is to make it easy to add new peephole optimizations to CompCert without increasing the compiler's trusted computing base. We believe that Peek's general approach could be applied in other verified compiler contexts without major changes.

Similar to CompCert, the Vellvm framework [30, 31] strives to provide a highly reliable suite of verified compilation tools. Alive [13] is a tool for verifying peephole optimizations in LLVM. Using Alive, a peephole writer can express and automatically prove their optimizations correct and extract the optimization to efficient C++ code which runs in the LLVM framework. The authors applied Alive to identify several bugs in LLVM's existing peephole optimizations. Unlike Alive, Peek requires peephole writers to manually verify their optimizations (though Peek provides extensive tactic support to make the proof burden reasonable). Peek is also proven correct in Coq and does not include the correctness of any constraint solvers in its TCB.

Infinite to Finite Memory Model Translation. We are not the first to formalize the translation of pointers from high-level infinite memory models to low-level finite memory models. Kang et. al. built and verified what they call a quasi-concrete memory model, which addresses the gap between infinite and finite memory models [7]. Their model injects high level pointers to low level bits on demand, as those pointers are cast to integers. If a pointer is never cast, it is never injected. Further, they give semantics to their injection (or as they call it, concretization), whereas we leave the specification for the allocator opaque. Recent work to develop a concrete memory allocator for CompCert [3] verifies translations against a simple, conservative memory allocator which lacks the ability to reuse memory. Our memory allocator interface allows for memory allocators that can reuse previously freed memory. The CompCertTSO [27] project built finite memory into every level of CompCert, dealing with many similar issues to us, but never explicitly translating from infinite to finite memory. This translation required our novel memory-allocator axiomatization and an equivalence theorem up to the first allocation failure.

8. Conclusion

We presented Peek, a framework for expressing meaning-preserving x86 program transformations in the CompCert verified compiler. To do so required (1) a new lower-level semantics in which pointers are 32 bits, (2) an axiomatization of dynamic memory management for proving the new semantics equivalent to the previous one under reasonable assumptions, (3) a verified assembly-level liveness analysis, assuming the standard calling convention, (4) a verified program-transformation engine for applying rewrites, (5) an interface for expressing peephole optimizations and proving them locally correct, and (6) a proof that locally correct rewrites are globally correct. Beyond all the necessary proofs, we evaluated our work by implementing and proving correct a variety of peephole optimizations.

Future work includes extending CompCert's x86 semantics to include more instructions and addressing modes that enable optimizations, and to investigate other assembly-level transformations such as software-fault isolation.

References

[1] A. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems*, 37(2):7:1–7:31, Apr. 2015.

[2] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 394–403, Oct. 2006.

[3] F. Besson, S. Blazy, and P. Wilke. A concrete memory model for CompCert. In *Proceedings of the 6th International Conference on Interactive Theorem Proving*, pages 67–83, July 2015.

[4] S. Buchwald. Optgen: A generator for local optimizations. In *Proceedings of the 24th International Conference on Com-*

piler Construction, pages 171–189, Apr. 2015.

[5] A. Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages*, pages 93–106, Jan. 2011.

[6] S. Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, Feb. 2005.

[7] J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis. A formal C memory model supporting integer-pointer casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–335, June 2015.

[8] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

[9] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226, June 2014.

[10] S. Lerner, T. Millstein, and C. Chambers. Cobalt: A language for writing provably-sound compiler optimizations. *Electronic Notes in Theoretical Compututer Science*, 132:5–17, 2005.

[11] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, pages 364–377, Jan. 2005.

[12] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[13] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–32, June 2015.

[14] H. Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–126, Oct. 1987.

[15] W. M. McKeeman. Peephole optimization. *Commun. ACM*, 8 (7):443–444, July 1965.

[16] J. S. Moore. A mechanically verified language implementation. *J. Autom. Reasoning*, 5(4):461–492, 1989.

[17] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. Rocksalt: Better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 395–404, June 2012.

[18] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: X86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 391–407, Aug. 2009.

[19] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 396–407, June 2014.

[20] S. Rideau and X. Leroy. Validating register allocation and spilling. In *Proceedings of the 19th International Conference on Compiler Construction*, pages 224–243, Apr. 2010.

[21] V. Robert and X. Leroy. A formally verified alias analysis. In *Proceedings of the 2nd International Conference on Certified Programs and Proofs*, pages 11–26, Dec. 2012.

[22] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 305–316, Mar. 2013.

[23] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 53–64, June 2014.

[24] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 111–121, June 2010.

[25] J.-B. Tristan and X. Leroy. Verified validation of lazy code motion. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 316–326, June 2009.

[26] J.-B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages*, pages 83–92, Jan. 2011.

[27] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), June 2013.

[28] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, Nov. 1997.

[29] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, June 2011.

[30] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th ACM Symposium on Principles of Programming Languages*, pages 427–440, Jan. 2012.

[31] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 175–186, June 2013.

A. Peephole Optimizations

Concrete register names are used for clarity, whereas actual verified transformations are parameterized, except in the case of instructions that require specific registers (e.g. division). Trailing `nop` instructions are frequently left off. The first six peepholes are from Bansal et al.'s Superoptimization paper [2].

```
1)  sub %eax, %ecx      notl %eax
    mov %ecx, %eax ==> add %ecx, %eax
    dec %eax

2)  test %ecx, %ecx     test %ecx, %ecx
    je .L0          ==> cmovene %edx %ebx
    mov %edx, %ebx
    .L0

3)  mov %eax, %ecx      xchg %eax, %edx
    mov %edx, %eax  ==>
    mov %ecx, %edx

4)  setg %al             mov $0, %eax
    movzbl %al, %eax ==> cmovg %eax, %esi
    dec %eax
    and %eax, %esi

5)  mov $8, %eax        mov $7, %eax
    sub %ecx, %eax ==> sub %ecx, %eax
    dec %eax

6)  mov %eax, -20(%ebp)     mov %eax, -20(%ebp)
    mov -20(%ebp), %ecx ==> mov %eax, %ecx

7)  add $1, %eax ==> inc %eax

8)  add $-1, %eax ==> dec %eax

9)  mov $2, %ecx ==> shr $1, %eax
    div %ecx

10) shr $12, %ebx
    shld $20, %ecx, %edx
    sal $20, %eax
    or %ebx, %eax

           ==> shld $20, %ecx, %edx
               shld $20, %ebx, %eax

11) sal $20, %eax  ==>  shld $20, %ebx, %eax
    shr $12, %ebx
    or %ebx, %eax

12) inc %eax  ==> nop
    dec %eax

13) inc %eax ==> inc %eax
    inc %eax
    dec %eax

14) jmp .L0 ==> nop
    .L0

15) lea (%eax,%ebx), %eax ==> add %ebx, %eax

16) mov $0, %eax ==> xor %eax, %eax

17) mov %eax, %eax ==> nop
    mov %ebx, %ebx

18) imul $2, %eax ==> sal $1 %eax

19) imul $4, %eax ==> sal $2 %eax

20) imul $8, %eax ==> sal $3 %eax

21) not %eax        ==>  sub %ebx, %eax
    add %ebx, %eax       not %eax

22) mov %eax, %ebx  ==>  mov %ecx, %ebx
    mov %ecx, %ebx

23) sub $1, %eax  ==> dec %eax

24) sub $-1, %eax ==> inc %eax

25) mov %eax, %ebx           test %eax, %eax
    lea -1(%ebx), %eax  ==>  lea -1(%ebx), %eax
    test %ebx, %ebx

26) xor %eax, %eax     ==> nop
    mov %eax, 24(%esp)     mov $0, 24(%esp)

27) lea (%eax,%ebx), %ebx ==> add %eax, %ebx

28) mov %eax, 56(%esp)
    mov 56(%esp), %edx
    and $15, %edx
    mov 0(%ecx,%edx,4), %eax

                 mov %eax, 56(%esp)
            ==>  and $15, %eax
                 mov 0(%ecx,%eax,4), %eax
                 nop
```