Peek: A Formally Verified Peephole Optimization Framework for x86

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

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

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

Abstract

Peek is a first step toward adding support for assembly-level program analyses, transformations, and optimizations in CompCert. Currently, Peek focuses on x86 peephole transformations implemented and verified in Coq. Peek is designed to provide a modular interface requiring that each peephole optimization satisfy only *local* correctness properties. Our primary result establishes that, assuming the C calling convention, any peephole optimization satisfying these local properties preserves *global* program meaning.

1. Overview

The CompCert C compiler [2] has consistently demonstrated incredible reliability in practice [1, 4] due to its end-to-end, machine-checked correctness guarantee. This guarantee ensures that any output assembly program will have the same behavior as the corresponding input source program. However, because each transformation must be be proven correct in full formal detail, CompCert currently lacks many common optimizations, especially for assembly level programs. As a consequence, assembly programs produced by CompCert are "frozen": any unverified modification to such programs invalidates the strong guarantees CompCert provides.

Low-level transformations are challenging because they must account for machine-level details such as status flags, the calling convention, and potential program-counter overflow. Furthermore, low-level transformations cannot rely on later compiler passes to clean up, and instruction addresses used in control flow are stored in the same memory accessed by arbitrary read and write operations.

To explore low-level transformations in CompCert, we are developing Peek, a framework for reasoning about analyses and optimizations over x86 programs. In this work, we briefly describe the major components of Peek and how they can be composed to support an extensible peephole transformation system in CompCert. Ultimately, we hope to use Peek to build many useful machine-level extensions such as SIMD support, software fault isolation, worst-case execution time guarantees, and modern superoptimizer support. However, while Peek provides a first step in many of these domains, there are still substantial challenges to address.

Peek comprises four major components: a liveness analysis, a symbolic evaluator, a representation for peephole optimizations and their local proofs, and a compiler pass to run such verified peephole optimizations. We discuss the design tradeoffs and verification challenges of each component, and report on insights gained to mitigate the proof burden.

Liveness Analysis. Consider the simple peephole that changes an `addl` instruction into `leal`. This is useful because it can increase instruction level parallelism, as `addl` is executed in the ALU while `leal` is executed by the memory unit. While these two instructions make identical changes to all normal program registers, they do *not* make identical changes to the CPU flags. Thus if an instruction reads the flags immediately after this transformation and branches on flag values, the transformation could change program behavior and thus be incorrect. However, if we require every peephole to leave *all* registers in identical states, almost no traditional peephole transformations will be verifiable.

Instead of requiring full state equality, Peek requires only that peepholes preserve the values of *live* registers. A live register is any register that could contain a value that might influence program behavior. To support this notion of peephole correctness, Peek provides an x86-level liveness analysis over registers.

We built and verified a standard iterative liveness analysis over CompCert x86 assembly, which produces a function from code locations to sets of registers. Due to the representation of control flow, we assume properties about the calling convention, such as where calls and returns step, and what registers are live across calls and returns. We define correctness of the liveness analysis as a relation on states that preserves program behavior. That is, liveness is correct if any two states that agree on live locations will produce the same program behavior.

Rewrite. Traditionally, a peephole rewrite is a find pattern and a replace pattern. The compiler searches out the find pattern, and replaces it with the replace pattern [3]. Modern compilers typically include a host of peephole optimizations, as there are many architectural details that are only exploitable at the assembly level. For example, comparison elimination, or using the memory unit to perform integer addition using `leal` are examples of common peephole optimizations that are simply not possible since the details that allow them are both architecture dependent and hidden at higher levels in the compiler.

In a verified compiler, we must prove that replacing the find pattern with the replacement code does not change program behavior. In Peek, a rewrite is represented as a dependent record. Currently, the `find` and `replace` patterns are simply lists of concrete instructions. Similar to traditional peephole rewrites, these specify the code to find, and the code to replace found code with. Liveness restrictions in Peek encode assumptions about liveness of registers at program locations where the rewrite will be valid. These liveness restrictions are represented by fields `live_in`, `live_out`, and `pres`, which are simply lists of registers assumed to be live on input, guaranteed to be live on output, and "don't care" respectively. The first two are liveness information for the rewrite, and the third is a set of registers that neither list of instructions modifies. The rewrite record also includes a measure, accompanied by a proof that it decreases with every step of execution within the find pattern. This guarantees that any execution does not diverge within the peephole. The rest of the fields are proofs about the data, from simple facts such as that neither list of instructions is empty, or that they have the same length[1], to richer properties, such as execution through either list of instructions preserves the values stored in the registers in `pres` or that execution through the two different lists of instructions produces the same values in the `live_out` registers, given identical starting values in the `live_in` registers.

Symbolic Evaluation. As mentioned earlier, Peek users must prove that the `find` and `replace` patterns preserve program behavior when used as a rewrite rule. We give users the ability to symbolically evaluate snippets of code, in order to let them prove that two snippets produce the same values for all possible executions. We expose two main functions: `sym_exec`, which takes a piece of code and produces a symbolic state, and `sstate_inst`, which takes a symbolic state and a concrete state, and instantiates the symbolic state with the concrete state as a seed. The main correctness theorem we prove is correspondence of concrete program execution with symbolic execution followed by instantiation.

In addition, we provide functionality to reason about execution of instructions that can trap. For example, integer division will raise an exception if the divisor is zero. We provide a notion of preconditions on concrete program states, which hold if and only if a state is able to step through a particular piece of code. In addition to making the user prove that the find and the replace pattern generate the same values in the same live registers, we force the user to prove that the precondition for the find pattern executing to completion holding implies that the precondition for the replace pattern holds.

Currently our symbolic evaluator supports only non-jump instructions, and does not support any instructions which access memory. The extension to support memory operations will be easy, but extending to support jumps will require significant engineering effort. However, most traditional peephole optimizations are over straight-line code.

Rewrite Engine. Peek provides a *rewrite engine*, a compiler pass to perform a single peephole rewrite on code as it is being compiled. Peek matches the find pattern of the rewrite code, checks if the location is a valid location to perform the rewrite, and performs the rewrite.

A potential rewrite location is a position in the code where we've found the list of instructions we are looking for, and we are considering replacing it with our list of instructions we replace things with. However, we need to know a few more facts about this location before we can perform the rewrite. We must know that the set of labels that occur in jumps and labels outside the rewritten region are entirely distinct from the set of labels that occur within jumps and labels within the find pattern and the replace pattern. Finally we must make 3 checks with the calculated liveness information for the function we are rewriting within:

1. the `live_in` set must be a subset of the actual liveness information calculated for the beginning of the potentially rewritten region.
2. the actual liveness for the exit of the potentially rewritten region must be a subset of the union of the `live_out` and `pres` sets.
3. Finally, if a register is in the `pres` set, it must be either marked live at both the entry and exit, or at neither the entry nor the exit.

If a location satisfies all of these criteria, it is a proper rewrite location.

The main proof of correctness for the rewrite engine is the bisimulation correspondence proof between the original program and the transformed program. It follows in a style similar to other parts of CompCert. Our `match_states` relation has 4 constructors, `outside`, `entry`, `inside`, and `external`, corresponding to states outside the transformed region (or in a function which wasn't transformed), states at the entry of a transformed region, states inside (not at the entry of) a transformed region, and states within an external function.

[1] In practice, the same length restriction can be subverted by inserting or removing `nop` instructions

Ongoing Work. One of the biggest challenges, establishing that an assembly program adheres to the C calling convention, is still unverified. This includes two separate kinds of facts about calls and returns. First, that all return instructions step to the instruction after a call, and all calls step to the beginning of a function. Second, that the callee save registers are the correct liveness information at a call instruction, and the callee save registers along with any result register(s) are the correct liveness information at a return instruction. Note that the calling convention *is* obeyed for all code generated by Compcert, but expressing the x86-level invariants and checking that they hold for an x86 program is non-trivial, but a necessary (and often unstated) correctness criterion for peephole optimization. For example, one function invocation doesn't access another's stack frame. We currently state but admit this sort of information.

Rewrites also must currently specify precisely the operand registers for each instruction: e.g. we look for `addl %eax, %edx`, not for any `addl` instruction. In the future we hope to parameterize these and extend the matcher, thus making each transformation apply more broadly.

Throughout CompCert, there are quite a few languages. The semantics for every langauge within CompCert is defined over the same values. They are typed values, which are either integers, longs, pointers, floating point, or undefined. While this is a convenient representation for reasoning over C semantics, a bit-based representation at the assembly level would be useful for two reasons. First, it would more accurately reflect what the processor actually does, and thus we could have greater confidence in its correctness. Second, we would be able to verify more peephole optimizations if in the semantics, assembly instructions computed over untyped bits, instead of typed values. Currently, Peek cannot turn integer addition of 0 to a register into a `nop`, since integer addition of 0 to a floating point value is undef, whereas `nop` leaves the floating point value untouched.

The current CompCert x86 semantics is sound but imprecise – as uncomfortable as it is to "change the semantics" as we grow the set of peephole optimizations, not doing so is limiting optimization opportunities that are completely valid

Preliminary Results. We are just starting to implement a set of peephole optimizations and apply them to benchmarks. On one of our microbenchmarks, replacement of two `leal` instructions with `addl` instructions yielded a speedup of 1.76x for this benchmark. Currently, our CompCert pass to apply rewrites does not significantly impact CompCert runtime, though we have only run the pass with a small small set of peepholes to date.

References

[1] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. PLDI '14, 2014.

[2] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. POPL '06, 2006.

[3] W. M. McKeeman. Peephole optimization. *Commun. ACM*, July 1965.

[4] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. PLDI '11, 2011.