Solidity Compiler

Modern Peephole Patterns Optimization

Philippe Laporte

savin' y'all some dough

Blockchain Technologies 6115
Information Systems Engineering
Concordia University

December 2 2025

In memory of Laurie Hendren

Preamble

Once upon a time I took a Compiler Design course with a Java Virtual Machine [1] theme. We learned all about code generation and early optimization for a subset of Java (JOOS) producing JVM class files, using the Jasmin [2] assembly language. Those were the days before the “all purpose Java bytecode manipulation and analysis framework“ ASM [3] came about. The optimizer included Peephole Patterns [4] whose application was driven by a fixed-point engine. The Java Virtual Machine features a stack-based execution engine.

The concept carries over to any architecture. A fixed-point driven peephole optimizer walks a sliding window of stride 1 along the previously generated code and tries to apply a list of transformations that check if a specific pattern of byte codes appear that have a known simplification, and terminates when none of the transformation functions find patterns that apply. Clearly the sequences in which the pattern simplifications are applied do not commute, hence why several iteration are required before the transformation process stabilizes.

My old notes also promise that even though each peephole pattern does not necessarily make the code smaller, the process does eventually terminate.

The testing strategy for each pattern then proposed three phases:
1- A careful argumentation that each peephole pattern is sound
2- A demonstration that each pattern is realized correctly
3- A statistics showing that the optimizer improves the generated byte code

We shall attempt to follow this methodology as we explore Peephole patterns optimization in the Solidity Compiler.

A few additional definitions

Solidity is the standard language for programming Smart Contracts that run on the Ethereum blockchain [5]. There is a standard compiler for it that runs as solc [6]. This compiler targets the Ethereum Virtual Machine (EVM) [7].

The EVM is a stack-based, big-endian VM with a word size of 256-bits and is used to run the smart contracts on the Ethereum blockchain. Smart contracts are just like regular accounts, except they run EVM bytecodes when receiving a transaction, allowing them to perform calculations and further transactions. An example instruction is PUSH to push a value on the top of the stack.

Both projects are Open Source meaning the general public can inspect and improve on the existing code as well as build and test it for themselves.

Methodology

Starting with no knowledge of Solidity or the Ethereum Virtual Machine, I ramped up on those. I went thorough the entire solidity specs [8] so as to be able to examine as many peephole patterns and their source code origins as possible. Indeed we seek to understand which source code patterns generate the code that can be optimized. I then went through the EVM bytecode specs [9] and discovered as well that contracts also can use inline assembly [10], a low-level form halfway between the source High-Level language and the byte code, as well as a higher-level and newer format of inline assembly called Yul [11]. We could also be interested in optimizing the code generated from this inline assembly. I chose to skip this aspect for the time being as detailed later.

I then looked into how relevant the standard Solidity Compiler's optimizations are in the context of the potential presence of a JIT Compiler "peer" in the EVM runtime. I found that the popular Open-source EVM runtime does not have a JIT component, but less widespread, proprietary implementations such as Monad do [12], as well as smaller Open Source projects such as Go-Ethereum [13]. Therefore Solidity compiler optimizations are most relevant and will remain so in the foreseeable future.

It is instinctive that the Solidity optimizer must aim to reduce both deployment and execution bytecode size and gas consumption. Let me explore this topic shortly.

Whereas the JOOS peephole optimizer examined the call graph, the peephole optimizer in the standard Solidity compiler is part of much more sophisticated optimization pipeline [14] in which most call graph analysis in done in another module. More on this below.

The code for the peephole optimizer resides in libevasm/PeepholeOptimiser.h/.cpp [15]. It is called into action by libevasm/Assembly.h/.cpp [16].

Conceptually moving fast-forward to today, LLMs are enhancing web search in a way I dreamed of when speaking with my adviser way back when as an undergraduate. I hereby claim there is no better way to go about this large project than using LLMs for code exploration and comprehension. The trick is not to let the LLM think for us, but rather, to let it answer targeted questions generated by the researcher. We are effectively turbo-charging the classical web searches. I skimmed through the entire Optimizer code but only needed to go through from start to finish the 1000 lines of the PeepholeOptimiser class. I informed myself regarding the rest of the code base using targeted LLM prompts.

Never trust the AI assistant blindly, however. Early in my research I asked if there is a fixed-point driver in the Solidity peephole optimizer and it said no, thus mis-directing my early research.

Indeed the LLM needs more context before it can answer satisfactorily, and just asking such a question out-of-the-blue so-to-speak is asking too much. I had to tell the LLM in which file the Optimizer resides. Then the fixed-point driver code is not in this file but rather in the Assembly class to which I will return later.

Seeing that the peephole optimizer already incorporates a Fixed-Point Driver focused my research on any improvement to this driver and on attempting to come up with new peephole patterns.

The Peephole Optimizer's place in the compiler's optimization pipeline

The optimizer architecture has two layers, Yul and Peephole. Since version 0.6.x, almost all real optimizations happen in the Yul optimization phase. The peephole optimizer is now basically the thing that cleans up stack-level noise that's too low-level for Yul to see. But the development team assumed rules wouldn't depend on each other, which is false today. Thus today the peephole optimizer:

- has more rules
- operates on noisier stack code

- exposes chain reactions

The compiler's `ExpressionSimplifier` already handles many potential patterns, but peephole rules can catch complex sequences generated by higher-level compiler passes that algebraic simplifications might miss.

The dedicated Peephole Optimizer is a final pass that runs after the code has been fully generated into EVM assembly/bytecode.

Our target measures of success: bytecode size AND gas consumption

Gas [17] refers to the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network. EVM has two separate bytecode worlds with different cost models:

1. *Runtime bytecode* (the code executed after deployment)

Most important concern: gas consumption

This is the code that runs on every call. It affects transaction cost and the user's fees. It affects your contract's competitiveness. It affects how much it costs for external actors to use your contract.

For runtime code, gas efficiency is king.

- Peephole patterns that reduce the number of EVM instructions and stack ops
- Removal of redundant work (DUPx POP, AND chains, useless arithmetic)
- Cheaper operations (no unnecessary SLOADs, no unnecessary jumps)

Bytecode size matters less here

2. *Creation bytecode* (constructor + deployment code)

Most important concern: bytecode size

Deployment cost is vastly dominated by the size of the creation bytecode. It is charged at 200 gas per byte, which is much more expensive than normal runtime execution. Constructor logic only executes once, so its runtime gas use barely matters.

For creation code, bytecode size is king.

- Peephole patterns that reduce PUSH sequences
- Removal of dead code and unused literals
- Minimizing large static arrays or metadata
- Shrinking constructor ABI unpacking logic

Gas cost of constructor execution is relatively trivial compared to the cost of storing the code itself.

3. *Runtime code size still matters* (gas refund / EIP changes)

Even though gas usage is more important for runtime code:

Large contracts face the EIP-170 [18] contract size limit (24 KB). Chains charge significantly based on calldata + bytecode size. Proxy patterns often rely on upgradeable code size constraints. So bytecode size of the runtime code does matter, but less than gas efficiency unless you're near limits.

Therefore the priority is:

If you're optimizing runtime code: Gas consumption dominates.

If you're optimizing creation code: Bytecode size dominates.

If your contract is close to the 24KB limit: Bytecode size may be more urgent for runtime too.

The Fixed-Point Driver

As mentioned earlier the fixed point driver can be shown to converge, yet the Solidity compiler code increments a counter on each loop and halts with an OptimizerException is that count reaches an arbitrarily large number (64k, which is outlandish). Looking at the commit message and related PR in Github from 2017 [19], the change was not focused on this counter but piggy-backed onto another mechanism to try and make the optimizer stop its work. It asks specifically:

Q: Do you have a test case which triggered a[n infinite] loop?

A: No, but the two conditions (number of pops and bytecode size) could oscillate.

This clearly appears vague and not rigorous. I thus offer a formal, Agent-generated proof in Appendix A that can back an eventual short and sweet PR to get rid of the count and let it loop forever with confidence that it will converge. The goal is indeed to show whether the fixed-point driver always converges in a bounded number of steps, or if a cycle is possible.

Investigating new Peephole patterns

As confirmed by the Superoptimizer paper [20], AI is best suited at finding patterns in general, and peephole patterns in particular. The investigation used hundreds of contract examples to deduce tons of useful patterns. I started with one simple Contract, compiled it with full optimization (both the traditional `–optimize` pipeline and the up-and-coming `–with-ir –optimize-yul` pipeline), fed the resulting opcodes (`--opcodes` flag) to the LLM and asked it to find peepholes, indeed, patterns that can be improved according to a set rule while respecting language semantics.

It was important to work this way, as merely asking the LLM for candidate patterns in general yielded a slew of false positives that were turned away upon a single compilation with `–optimize –via-ir –optimize-yul`, asking the LLM to find those patterns and confirming that they do not actually occur in the suggested contract code. So this was an unsuccessful early heuristic. The LLM had to be trained that it only mattered to look at fully optimized code to find new optimization opportunities. It also got side-tracked looking at Yul generation instead of bytecodes.

The Agent ultimately generated the source code for each added pattern as well as the unit tests, with several rounds of tuning feedback on failing semantic test.

Engineering New Patterns

Git-blame reveals activity in the last year in the PeepholeOptimiser code thus it surely is not dead-code from that perspective. I also found no issue where the maintainers explicitly reject expanding the peephole optimizer. The issues that *do* exist around optimizer behavior are adjacent (defaulting to Yul for example) but not directly dealing with adding new peephole rules. Some issues are marked “selected for development” but that tag refers to Yul optimization settings, not clearly to the opcode peephole expansion.

Thus proceeding to propose a patch and initiating a pull request seems reasonable and unlikely to conflict with an existing initiative.

Should we take into account the `--optimize-run` command-line option which allows the user to put an emphasis either on contract deployment or recurring execution? The current design philosophy appears to be that peephole optimizations should always be win-win (smaller code AND faster execution), so they don't need to consider the trade-off that `–optimize-run` represents.

The Solidity Team’s PR acceptance likelihood

The Solidity team does accept Prs for optimizer improvements when they are:

- correct, fully tested and shown to be safe

But almost no one has touched significantly the peephole optimizer in years. The opcode-level peephole pass is one of the least-evolved parts of the compiler. It has a very small rule set. This area is certainly underdeveloped compared to the Yul efforts. So there is indeed plenty of work to be done, and very few people are working on this specific part.

Optimizing for Gas usage

The current code considers only reduction in byte code to determine whether a pattern is successful. The pattern I worked on that was going to be applied to inline assembly patterns such as

```
pragma solidity ^0.8.20;
contract TriggerMul32 {
    function mul32(uint x) external pure returns (uint) {
        uint y;
        assembly {
            // Manually force the pattern PUSH1 0x20 ; MUL
            // (x * 32)
            y := mul(x, 0x20)
        }
        return y;
    }
}
```

does not reduce the bytecode size but does reduces the gas consumption. Indeed the bytecode transformation is

PUSH <2^k>
MUL
→

PUSH <k>
SHL

This has the same number of byte codes but does not consume the same amount of gas. This is due to the rules of the PeepholeOptimizer for specific pattern application as the new code must have fewer items, or:

- If the same number of items:
- It must require fewer bytes, or
- It must have more POPs (a proxy for “we got rid of useless results”).

Our MUL → SHL peephole has the same number of items (2 vs 2), has no extra POPs and only reduces byte size if the original constant needs more than 1 byte (i.e., PUSH2, PUSH3, etc). For 32 (0x20), the original is PUSH1 0x20, which is already minimal:

- Before: PUSH1 0x20 (2 bytes) + MUL (1 byte) = 3 bytes
- After: PUSH1 0x05 (2 bytes) + SHL (1 byte) = 3 bytes

So for 32, the optimizer says:

> “No size improvement, no extra POPs → I’m not going to keep your change.”

For a larger constant like 1024 (0x0400), we *do* win:

- Before: PUSH2 0x0400 (3 bytes) + MUL (1 byte) = 4 bytes
- After: PUSH1 0x0a (2 bytes) + SHL (1 byte) = 3 bytes

Should we thus restrict the pattern to fit the framework, or should we change the framework to take in gas consumption reductions? We certainly don’t want to have pattern-specific mechanisms such as:

Accept optimized code sequence if bytes got smaller OR containsMulToShlRewrite()

Rather, we’d like the optimizer to take into account gas usage reductions. Patterns can then assess the gas usage benefit by using known constants, for our example above

Gas Benefit = GAS_MUL - GAS_SHL;

Values come from EIP-150 [21] and Berlin [22] gas tables indeed indicate:

MUL costs ~5 gas
SHL costs ~3 gas
Benefit ≈ 2 gas

So should we introduce such an improvement? Adding gas-price-based heuristics could be:

- Inconsistent across hard forks

- Brittle
- Hard to maintain

Thus, feasible but potentially challenging.

The current situation

Currently, gas reduction is done only incidentally, not systematically. The docs [8] repeatedly emphasize that the Solidity optimizer is a size optimizer, not a gas optimizer. Specific quotes:

“The optimizer mainly aims to reduce the bytecode size.”
→ This is the core design.

“Gas savings are not explicitly optimized for.”
→ Mentioned indirectly but clear from the description.

“Transformations are only applied if the generated code is shorter.”
→ Confirms byte-size is the acceptance criterion.

“Only instructions that reduce bytecode size are applied automatically.”
→ Confirms peephole is size-only.

There is no gas-cost-driven optimizer anywhere in the codebase. The only components that consider gas at all:

- Yul Optimizer (but still size-focused)
- Common Subexpression Eliminator (reduces gas indirectly)
- Constant Optimizer (SSA) (may reduce runtime ops)
- Inlining (reduces jumps but can increase size; compiler avoids doing it unless size also reduces)

But in every case, Gas is a secondary side effect, is never the approval metric and bytecode size is always the deciding factor. Thus we may move gas-aware transforms to the Yul optimizer instead.

This is what the Solidity team would approve of most easily. Yul’s optimizer already has a richer cost model and is “semantic level” so it can reason about replacement safety. We would add a pass like:

```
ArithmeticStrengthReductionPass
```

that transforms:

- mul(x, 2^n) → shl(n, x)
- div(x, 2^n) → shr(n, x)
- mod(x, 2^n) → and(x, 2^n - 1)

This follows the model of the existing pattern replacements.

Up side:
- Compiler team likely to approve
- Safer

- Generalizable
- Better architecture

Down side:
- Requires some plumbing in Yul IR.

So we are looking at another PR with this improvement!

Newly introduced Patterns

Methodology

For each new pattern identified by the Agent in the supplied bytecodes, we generate the source code to hook in the pattern detection and substitution as well as the unit test. We build the new compiler and the run the unit test with the command

```
./scripts/soltest.sh --no-semantic-tests –no-smt
```

and make sure all pass

then we comment out the new pattern, rebuild solc with –opcodes –optimize as such

```
solc --overwrite --bin --optimize pattern_example.sol -o before
```

and we get a gas consumption report from running

```
evmc run --vm ~/Sources/evmone/build/lib/libevmone.so @before/Example.bin
```

which runs the code on the standard EVM engine

We get a gas consumption report such as

```
Gas used: 97
```

and confirm the improvement

Then in preparation for the PR [23], since our PR affects gas costs we run

```
isoltest --enforce-gas-cost --optimize –accept-updates
```

Here we hit a failure!

The semantic tests revealed 1 of the 3 patterns identified earlier as unsound, and so it was removed.

Then we saw that our sound pattern triggered test expectation fixes on the spent gas, a sure sign of success.

Our 2 Patterns

We describe two local peephole optimization patterns implemented in the Solidity EVM code generator. Both target common instruction sequences generated by the compiler and replace them with semantically equivalent, but more efficient forms. Or so did the Agent think…

Peephole Optimization Pattern #1: FunctionSelectorGuard

The FunctionSelectorGuard peephole optimization targets the Solidity-generated dispatcher guard that ensures at least 4 bytes of call data are available for reading the function selector. This optimization is safe, semantics-preserving, and reduces both bytecode size and execution gas.

Before (unoptimized dispatcher)

```
JUMPDEST
PUSH1 0x04        ; constant 4
CALLDATASIZE      ; cds
LT                ; cds < 4
ISZERO            ; cds >= 4
PUSH <tag>
JUMPI
```

After (optimized)

```
JUMPDEST
PUSH1 0x03        ; constant 3
CALLDATASIZE
GT                ; cds > 3  ≡  cds >= 4
PUSH <tag>
JUMPI
```

This rewrite is logically equivalent and safe. To guarantee correctness, the optimizer applies this rule only when the pattern immediately follows a Tag, ensuring it belongs to the actual function selector dispatcher.

Solidity Contract example that triggers the pattern:

```
pragma solidity ^0.8.20;

contract TriggerSelectorGuard {
    function foo(uint256 x) external pure returns (uint256) {
        return x + 1;
    }
}
```

Compiling with optimization produces a standard dispatcher containing the CALLDATASIZE ≥ 4 check, which matches the rewrite rule.

Bytecode Size Reduction

Opcode Sequence	Size
PUSH1 0x04, CALLDATASIZE, LT, ISZERO	5 bytes
PUSH1 0x03, CALLDATASIZE, GT	4 bytes

Net savings: 1 byte

Deployment gas cost is 200 gas per byte → ~200 gas saved.

Execution Gas Savings

Before

PUSH1	3 gas
CALLDATASIZE	2 gas
LT	3 gas
ISZERO	3 gas

Total: 11 gas

After

CALLDATASIZE	2 gas
PUSH1	3 gas
GT	3 gas

Total: 8 gas

Raw savings: 3 gas
Measured via evmone: 2 gas

Real-world dispatcher structure reduces the effective improvement by ~1 gas.

Summary of Optimization Benefits

- Safe (only applies after JUMPDEST)
- Semantically exact ((cds ≥ 4) ⇔ (cds > 3))
- ~200 deployment gas saved
- 2 execution gas saved per external call
- 1 byte smaller runtime code

This optimization improves nearly every contract compiled with Solidity!

A formal proof of soundness was generated at
https://chatgpt.com/s/t_692d192336bc819193cce87ee4229df3

Pattern #2: ErrorDispatcherStandardPanic

[Canonical Before/After Pattern skipped]

Bytecode Size and Gas Trade-offs

The ErrorDispatcherStandardPanic pattern is primarily an execution-path canonicalization and gas optimization for the panic case; it does not necessarily reduce deployed bytecode size.

Selector construction part only:

Before: PUSH4 (5 bytes) + PUSH1 (2) + SHL (1) + PUSH0 (1) + MSTORE (1) = 10 bytes.
After: PUSH32 (33 bytes) + PUSH0 (1) + MSTORE (1) = 35 bytes.

So the optimized form increases the number of bytes in the panic dispatcher by roughly 25 bytes, corresponding to about 5000 additional gas at deployment for that block. This is acceptable because panic paths are, by design, exceptional and are not expected to be taken during normal execution.
On the other hand, at runtime the optimized form executes fewer opcodes. At a high level, PUSH4 + PUSH1 + SHL (3 instructions) are replaced by a single PUSH32. Under the standard gas schedule, all PUSHx cost 3 gas, and SHL costs 3 gas, so the selector computation part goes from roughly 3 + 3 + 3 = 9 gas down to 3 gas, saving about 6 gas whenever the panic path is taken.
Because panic paths should be rare in correct programs, this is a classic space–time trade-off: slightly larger bytecode and higher deployment gas, in exchange for fewer instructions and lower gas when an error occurs. The canonical PUSH32 form also simplifies static analysis and pattern matching of panic revert payloads.

NOW wait a minute...**this makes no sense**: a code sequence which is unlikely to be executed at run-time surely does not warrant more expensive deployment all the time! The agent is trying to tell that a run-time saving of 6 gas is worth a deployment cost increase of 5000 gas!!!

So we reject this pattern and note that the Agent requires further training :-)

Our local optimization provides a modest but measurable improvement in deployment gas and execution gas across many contracts compiled by Solidity, while keeping the transformations simple, local, and easy to reason about.

Pull Request

The pull request for the added patter currently appears on my own repository fork [23] while awaiting review. I have logged an issue in the main repository [24] arguing for the change. The issue and the PR presentation were mostly generated by the AI agent with small edits of mine. I consulted the review checklist [25] and made sure to strictly follow it in order to maximize the chances of a quick review.

What’s next?

It is manifest that we can find more patterns using the same heuristic of presenting compiled contracts to the Large Language Model (LLM, AI agent) and asking it to find peephole patterns. Possibly using ever more Solidity contract source samples until a fixed point is reached so-to-speak.

Inline Assembly Patterns

According to a 2022 “Study of Inline Assembly in Solidity Smart Contracts” about 23% of analyzed deployed smart contracts (on Ethereum) contain inline-assembly (i.e. Yul/assembly) code [25]. Thus we must design peephole patterns that optimize ALL 3:

1. Yul-generated code
2. Solidity frontend–generated (non-Yul) assembly
3. User-inline assembly

Indeed solc has two completely different code generators. The peephole optimizer runs after both pipelines. It is literally the last line of defense before encoding the final bytecode. Meaning:

- ANY code that survives Yul optimization
- ANY inline assembly
- ANY fallback path
- ANY constructor code
- ANY metadata handling
- ANY optimizer escape hatch

All flow into the peephole pass. Thus Peephole patterns *cannot* be specialized only to one pipeline, as they have no idea which pipeline the instructions came from

The path forward: the Super-Optimizer

As we demonstrated how AI agents are particularly good at finding bytecode peephole patterns one may ask if we could train a model from scratch with a heuristic that combines Constraint Solving assisted by Supervised Learning and samples being code generated by the compiler while the targets being optimized patterns. This has been explored in [26].

This paper on super-optimization for the Solidity Compiler, titled “GASOL: GAS Optimization toolkit” reports great savings due to sound algorithms and heuristics. There is no doubt that it is an approach worth investigating further. We can ask whether the Super-Optimizer claims to replace only the peephole patterns phase or the entire optimization backend. In the POC paper they list convincing benchmarks comparing 3 configurations:

> *1. YUL+No-GASOL: It corresponds to the use of the YUL optimizer of solc compiler (−−optimize option).*
>
> *2. No-YUL+GASOL: It corresponds to the use of GASOL on the EVM bytecode generated by solc compiler when the Yul optimizer is disabled (−−optimize −−no−optimize −yul options).*
>
> *3. YUL+GASOL: It corresponds to the use of GASOL on the EVM bytecode generated by solc compiler when it is executed with the complete optimization options (−− optimize option).*

They then observe that:

> *Globally, if we consider all the data obtained from the analysis of the 30 contracts, we get that the gas savings are: 0,43% in the YUL+No-GASOL setting and 0,84% in the NoYUL+GASOL setting. Therefore, the gains obtained by GASOL are higher (double) than those obtained by the YUL optimizer when they are applied to the same EVM bytecodes (YUL+No-GASOL and No-YUL+GASOL blocks in Table 2.2). However, while the YUL optimizer is executed in a few seconds, GASOL needs much more time to optimize the contract. Interestingly, the gas savings of YUL+GASOL are 1,28%. Therefore, GASOL is able to optimize more gas when it is applied together with YUL (YUL+GASOL setting) than when it is applied alone (No-YUL+GASOL setting).*

Very promising indeed...recall that currently, solc features fewer than 20 rules compared to other compilers such as LLVM's 1000+ rules...who will take the relay? Indeed, why have those findings not resulted in new peephole patterns being merged into the solc code?

Super-Optimizer acceptance criteria

The findings have not been immediately or entirely merged into the main solc code due to several critical challenges inherent to compiler development for high-stakes environments like Ethereum.

The primary reasons are trust, formal verification, and the practical complexity of implementing novel optimization techniques.

Trust and Formal Verification (The security imperative)

In the context of blockchain smart contracts, correctness (security) is exponentially more important than efficiency (gas savings). A compiler bug introduced by an optimizer could lead to subtle, costly vulnerabilities.

- Lack of Trust in Unverified Tools: Traditional compilers are not formally verified, and super-optimizers, while mathematically sound in their search, are complex tools. Integrating their output, which consists of automatically generated, non-obvious rules, raises a high bar for trust.
- Need for Formal Proof: Before a new optimization rule is merged into solc, the Solidity team prefers that the rule's semantic equivalence (i.e., that the optimized code produces the exact same result as the original code for all possible inputs) be formally verified and proven correct. Manual verification is time-consuming, and automated generation of correct, production-ready proofs for thousands of rules is a significant challenge.

Implementation and Integration Complexity

- Rule Conflict and Confluence: Schett and Nagele's work [27] specifically found that their generated rules showed non-confluence. This means the final optimized output depends on the order in which the rules are applied, which can lead to unpredictable or suboptimal results.

Integrating such a set of rules requires complex mechanisms to resolve these ordering conflicts, which adds significant complexity to the compiler's codebase.

Computational Scalability (Super-Optimizers are slow)

While super-optimizers are great for discovering optimal sequences for small basic blocks, they cannot be run practically on every block of a large contract during a standard compilation process.

- The Peephole Solution: The goal of the research was to extract the fast, local, guaranteed-correct patterns and put them into the solc peephole optimizer, which is a fast, linear-time process. The slow discovery phase must be separated from the fast deployment phase.

In summary, the transition from an academic discovery tool (which prioritizes finding the theoretical optimum) to a production-ready compiler component (which prioritizes speed, trust, and zero risk) is a difficult, multi-step engineering process that requires careful integration into the existing compilation pipeline.

Hence why formal proofs are provided for our darling as I hope to get its PR merged

AI Agent Chats

Here we have the first exploratory chat which mis-directed me into thinking there was no pre-existing Fixed Point Driver:
https://chatgpt.com/share/69198d8c-01c8-8013-b3f1-a52448c55805

Here is the chat that led to the superoptimizer paper and presentation
https://gemini.google.com/share/19c9bf9c8df2

Here is the discussion on the occurrence of inline assembly
https://chatgpt.com/share/692d1361-2318-8013-9795-20a88feb5b3d

Here is the crux of the research and Q&A
https://chatgpt.com/share/692cbeef-9720-8013-b907-01e369ce16cf

Last, I asked Github pilot in VSCode to give a proof of convergence for the Fixed-Point Driver, only to find out quite unexpectedly that the chat session cannot be shared. This proof appears in appendix A.

APPENDIX A

Fixed-Point Driver Convergence Proof Outline (Agent-generated)

Let's proceed with a concrete proof attempt, using the actual list of peephole patterns implemented in your optimizer. The goal is to show whether the fixed-point driver always converges in a bounded number of steps, or if a cycle is possible.

1. List of Patterns

PushPop
OpPop
DoublePush
DoubleSwap
CommutativeSwap
SwapComparison
DupSwap
IsZeroIsZeroJumpI
EqIsZeroJumpI
DoubleJump
JumpToNext
TagConjunctions
TruthyAnd
UnreachableCode
(plus: IsZeroIsZeroRJumpI, EqIsZeroRJumpI, DoubleRJump, RJumpToNext, DeduplicateNextTagSize3/2/1, FunctionSelectorGuard, OpStop, OpReturnRevert, Identity, etc.)

2. Pattern Effects and Monotonicity

Let's analyze each for monotonicity (does it always reduce code size or complexity, and can it re-enable another pattern?):

PushPop: Removes a PUSH or DUP followed by a POP. Always reduces code size by 2. Cannot re-enable itself or others.
OpPop: Removes a pure operation whose result is immediately POPped, replacing with POPs for its arguments. Always reduces code size. Cannot re-enable itself or others.
DoublePush: Collapses two identical PUSHes into one plus a DUP. Reduces code size by 1. Cannot re-enable itself or others.
DoubleSwap: Collapses two identical SWAPs. Reduces code size by 1. Cannot re-enable itself or others.
CommutativeSwap: Reorders commutative operations for canonical form. May not reduce code size, but does not increase it. Could expose new opportunities for other patterns, but not cycles.
SwapComparison: Reorders comparison operations for canonical form. Same as above.
DupSwap: Collapses DUP followed by SWAP if possible. Reduces code size by 1. Cannot re-enable itself or others.
IsZeroIsZeroJumpI / IsZeroIsZeroRJumpI: Simplifies ISZERO(ISZERO(x)) before JUMPI/RJUMPI. Reduces code size by 1. Cannot re-enable itself or others.
EqIsZeroJumpI / EqIsZeroRJumpI: Simplifies EQ(x, 0) ISZERO before JUMPI/RJUMPI. Reduces code size by 1. Cannot re-enable itself or others.

DoubleJump / DoubleRJump: Collapses consecutive JUMPs. Reduces code size by 1. Cannot re-enable itself or others.
JumpToNext / RJumpToNext: Removes JUMP/JUMPI/RJUMP to the next instruction. Reduces code size by 1. Cannot re-enable itself or others.
TagConjunctions: Simplifies tag conjunctions. Reduces code size or leaves it unchanged. Cannot re-enable itself or others.
TruthyAnd: Simplifies AND with truthy values. Reduces code size or leaves it unchanged. Cannot re-enable itself or others.
UnreachableCode: Removes unreachable code after terminal instructions. Reduces code size. Cannot re-enable itself or others.
DeduplicateNextTagSize3/2/1: Deduplicates code after tags. Reduces code size. Cannot re-enable itself or others.
FunctionSelectorGuard, OpStop, OpReturnRevert: All are strictly reducing or canonicalizing, never increasing code size.
Identity: Always copies the current item if no other rule matches; does not change code.

3. Interaction and Cycles

All patterns either reduce code size or leave it unchanged.
No pattern increases code size.
No pattern can re-enable a previously removed sequence (e.g., none of these patterns can insert a PUSH+POP after it's removed).
Canonicalization patterns (CommutativeSwap, SwapComparison, etc.) may expose new opportunities for other patterns, but since they do not increase code size, and all other patterns are strictly reducing, the process cannot cycle indefinitely.

4. Bounded Steps

The only way the optimizer could loop forever is if a pattern could re-enable itself or another in a way that cycles, but with the above, this is not possible.
Each pass either reduces code size or leaves it unchanged, and the number of instructions is finite.
Therefore, the number of steps is bounded by the initial code length (or a small multiple, if canonicalization exposes new reductions).

5. Conclusion

Proof:
Given the current set of patterns, the peephole optimizer's fixed-point driver will always converge in at most as many steps as the initial code length (or a small multiple thereof, due to canonicalization). No cycles are possible, and the count guard is not strictly necessary for correctness, though it remains a good safety net for future pattern additions.

Summary:

The optimizer always terminates in a bounded number of steps.
The bound is proportional to the initial code length.
No infinite loops or cycles are possible with the current patterns.

References

[1] https://en.wikipedia.org/wiki/Java_virtual_machine
[2] https://github.com/davidar/jasmin
[3] https://asm.ow2.io/
[4] https://en.wikipedia.org/wiki/Peephole_optimization
[5] https://ethereum.org
[6] https://github.com/argotorg/solidity
[7] https://ethereum.org/developers/docs/evm/
[8] https://docs.soliditylang.org/en/v0.8.30/
[9] https://github.com/ethereumbook/ethereumbook/blob/develop/13evm.asciidoc
[10] https://docs.soliditylang.org/en/v0.8.30/assembly.html
[11] https://docs.soliditylang.org/en/v0.8.30/yul.html
[12] https://www.monad.xyz/
[13] https://blog.ethereum.org/2016/06/02/go-ethereums-jit-evm
[14] https://docs.soliditylang.org/en/v0.8.30/internals/optimizer.html
[15] https://github.com/argotorg/solidity/blob/develop/libevmasm/PeepholeOptimiser.h
[16] https://github.com/argotorg/solidity/blob/develop/libevmasm/Assembly.cpp#L903
[17] https://ethereum.org/developers/docs/gas/
[18] https://eips.ethereum.org/EIPS/eip-170
[19] https://github.com/argotorg/solidity/pull/3077
[20]https://drops.dagstuhl.de/storage/01oasics/oasics-vol084-fmbc2020/OASIcs.FMBC.2020.3/OASIcs.FMBC.2020.3.pdf
[21] https://eips.ethereum.org/EIPS/eip-150
[22] https://eips.ethereum.org/EIPS/eip-145
[23] https://github.com/argotorg/solidity/pull/16315
[24]https://github.com/argotorg/solidity/issues/16316
[25]https://github.com/argotorg/solidity/blob/develop/ReviewChecklist.md
[26] https://chaliasos.com/assets/papers/inline-oopsla22.pdf?utm_source=chatgpt.com
[27] https://github.com/costa-group/gasol-optimizer/blob/main/reports/gasol0.pdf and https://github.com/costa-group/gasol-optimizer/blob/main/reports/gasol1.pdf