apc-optimizer : A Verified Constraint-System Optimizer

Built 2026-07-31 20:40 UTC · commit aec6912

Contents

1 Introduction

This document describes `apc-optimizer`[1], a formally verified zkVM circuit optimizer. `apc-optimizer` is a core component in the powdr autoprecompiles[2] pipeline.

2 Background: zkVMs and autoprecompiles

zkVMs such as OpenVM[1], SP1[2], or powdr WASM[3] are virtual machines that output a cryptographic proof that a program executed correctly. They differ in the instruction sets they emulate but use the same underlying primitives: collections of *circuits* communicating via shared *buses*.

Each circuit is responsible for one or more instructions in the instruction set and is defined by a set of *constraints* over a *prime field*. Buses serve two purposes:

- They implement *lookups* into precomputed tables. For example, a byte range check might be implemented by proving that a circuit variable's value is in a size-256 table of all bytes.
- They implement *stateful communication* between circuits. For example, a *memory* is implemented via bus interactions.

Autoprecompiles prove correct execution of an entire *basic block*, which is a sequence of assembly instructions that can only be entered at the first instruction and exited at the last. The initial circuit is compiled from the instruction circuits of the zkVM and the concrete assembly program by instantiating whatever circuits would have been used by the vanilla zkVM within one monolithic circuit:

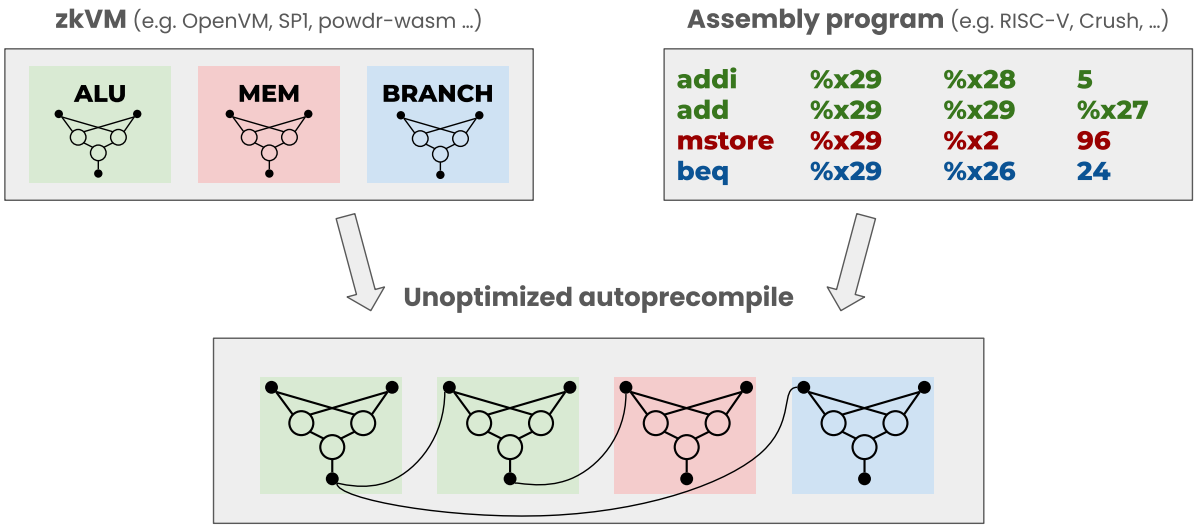
[1] powdr labs (2026). "Formally Verified Autoprecompiles" (`https://powdr.org/blog/formally-verified-autoprecompiles`). *powdr labs blog.* .

[2] powdr labs (2025). "Accelerating Ethereum with Autoprecompiles" (`https://powdr.org/blog/accelerating-ethereum-with-autoprecompiles`). *powdr labs blog.* .

[1] OpenVM Contributors (2026). "OpenVM Whitepaper" (`https://openvm.dev/whitepaper.pdf`). *Whitepaper.* .

[2] Succinct (2024). "SP1: a performant, open-source zkVM for RISC-V" (`https://github.com/succinctlabs/sp1`). *Software, https://github.com/succinctlabs/sp1.* .

[3] powdr labs (2026). "powdr-wasm: an optimized zkVM for WebAssembly" (`https://powdr.org/blog/powdr-wasm`). *powdr labs blog.* .



Having all instructions within the same circuit enables various optimizations, typically shrinking the circuit size by a factor of 3–4. Examples of these optimizations include:

- **Inlining of constants**. For example, immediate values can be inlined directly, as they are known at compile time. In combination with constant propagation, this specializes the circuit to the concrete basic block being proved.
- **Memory optimizations**. When proving instruction-by-instruction, even temporary values are written to memory and read back. Within a monolithic circuit, these values can be accessed directly, avoiding the overhead of the memory argument. One consequence is that each register is accessed only once, regardless of how many instructions read or write it.
- **Gadget optimizations**. Circuits often contain repeated subcircuits that can be optimized for how they are used. An example is RISC-V's `SEQZ` pseudo-instruction, which sets the output register to 1 if the input is zero, and 0 otherwise. It expands to the `SLTIU` instruction (a less-than comparison) with immediate value 1. But there exists a more efficient circuit for this specific comparison than the general-purpose `SLTIU` circuit.

In the remainder of this document, we formalize the properties that an optimizer must satisfy to be considered *correct*.

3 Variables, expressions and assignments

A *variable* is how the *runtime witness data* is referenced in a circuit. Variables in the input circuit carry a *powdr ID*, while newly introduced variables do not.

```
structure
Variable : Type
```

A circuit variable.

Constructor

`Variable.mk`

Fields

`name : String`

The display name of the variable.

`powdrId? : Option ℕ`

The optional powdr variable ID. All variables mentioned in the input circuit are expected to have a powdr ID. The output circuit may contain newly introduced variables whose values can be derived from a valid assignment of the input circuit.

An *expression* is defined inductively as a constant, a variable, or the sum or product of two expressions.

inductive type

```
Expression (p : ℕ) : Type
```

An arithmetic expression over structured variables and field constants.

Constructors

`| Expression.const {p : ℕ} (n : ZMod p) : Expression p`

A constant field element.

`| Expression.var {p : ℕ} (x : Variable) : Expression p`

A reference to a variable.

`| Expression.add {p : ℕ} (e1 e2 : Expression p) : Expression p`

The sum of two expressions.

`| Expression.mul {p : ℕ} (e1 e2 : Expression p) : Expression p`

The product of two expressions.

An *assignment* maps every variable to a concrete field value. An expression is *evaluated* under an assignment by folding its constants, variables, sums, and products into a single field element.

```
def Expression.eval (e : Expression p)
    (assignment : Variable → ZMod p) : ZMod p :=
  match e with
  | .const n => n
  | .var x => assignment x
  | .add e1 e2 => e1.eval assignment + e2.eval assignment
  | .mul e1 e2 => e1.eval assignment * e2.eval assignment
```

4 Bus interactions

A *bus interaction* sends a *payload* tuple to a bus, weighted by a *multiplicity*. Multiplicities are usually constrained to be 1 (a *bus send*), -1 (a *bus receive*), or 0 (no effect) in practice.

structure

```
BusInteraction (α : Type) : Type
```

A bus interaction. Typically, α is

- an expression (*symbolic bus interaction*), or
- a field element (*bus interaction message*).

Constructor

`BusInteraction.mk`

Fields

`busId` : $\mathbb{N}$

The ID of the bus this interaction is for. Distinct buses cannot interact.

`multiplicity` : `α`

The multiplicity with which the message is sent to the bus.

`payload` : `List α`

The payload of the bus interaction.

A circuit (defined below) contains a list of *symbolic bus interactions* (i.e., elements of type `BusInteraction Expression`). For a concrete run of the zkVM, the circuit might be instantiated several times with different variable assignments. Evaluating the symbolic bus interactions under an assignment yields a list of *bus messages* (i.e., elements of type `BusInteraction (ZMod p)`).

4.1 Bus state

The *bus state* of a circuit instance is the *net* effect it has on the buses, i.e., the net multiplicity each message is sent with:

```
/-- A concrete bus interaction message: which bus, and the tuple sent. -/
abbrev BusMessage (p : ℕ) := Nat × List (ZMod p)

/-- The effect on the stateful buses: the *net* multiplicity each message is
    sent with. -/
abbrev BusState (p : ℕ) := BusMessage p → ZMod p
```

Buses must *balance globally*: summed over all circuit instances in the entire zkVM execution, the net multiplicity of each message must be zero. This is enforced by the zkVM's proving backend, typically employing a protocol such as logup [1] [2].

4.2 Stateful and stateless buses

In practice, buses fall into one of two categories:

- A *stateless bus* or *lookup* is one where *most* circuits are constrained to only send messages with multiplicity 1 or 0. To balance it, a dedicated circuit *receives* messages with an unconstrained multiplicity. In this chip, the payload is fixed. Therefore, this implements a lookup: By sending to this bus, the prover proves that the sent payload is in the precommitted table.
- A *stateful bus* is one where the multiplicity can be 1 or -1 in any circuit. This implements a stateful communication channel between circuits. An example of this is the *execution bridge*: Each instruction chip might *receive* the current $(pc, timestamp)$ pair, and *send* the next $(pc', timestamp')$ pair.

As we will see below, we require that the optimizer preserves the net effect on stateful buses.

4.3 Memory

Most zkVMs implement a memory argument based on the *offline memory checking argument* due to Blum et al. [3]. They reduce memory consistency to a *multiset equality* check, which is essentially implemented by the bus argument.

In short, each read or write memory access is implemented as a series of bus interactions:

- An $(address, value, timestamp)$ pair is *received* from the memory bus.
- $timestamp$ is asserted to be *smaller than the current timestamp*. This is usually implemented via a limb decomposition and range checks via lookups.
- An updated $(address, value', timestamp')$ pair is *sent* to the memory bus, with $timestamp'$ equal to the current timestamp. Also, in the case of a *read access*, $value'$ is asserted to be equal to $value$.

In addition, there are circuits responsible for memory initialization and finalization. All in all, memory consistency is reduced to checking that the bus is balanced. Note that the send and receive directions might also be inverted.

As we will see below, we will assume that all circuits *including the circuit to be optimized* adhere to the memory discipline. If this was not the case, the original zkVM would not be sound in the first place.

5 Circuits

A *circuit* is simply a collection of algebraic constraints and symbolic bus interactions:

[1] Ulrich Haböck (2022). "Multivariate lookups based on logarithmic derivatives" (`https://eprint.iacr.org/2022/1530`). *Cryptology ePrint Archive, Paper 2022/1530.* .

[2] Shahar Papini and Ulrich Haböck (2023). "Improving logarithmic derivative lookups using GKR" (`https://eprint.iacr.org/2023/1284`). *Cryptology ePrint Archive, Paper 2023/1284.* .

[3] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor (1994). "Checking the Correctness of Memories" (`https://doi.org/10.1007/BF01185212`). *Algorithmica.* ***12****(2), pp. 225–244.*

structure

```
Circuit (p : ℕ) : Type
```

A circuit representing a single zkVM chip.

Constructor

`Circuit.mk`

Fields

`algebraicConstraints : List (Expression p)`

The list of algebraic constraints. For an assignment to be valid, all of them must evaluate to zero.

`busInteractions : List (BusInteraction (Expression p))`

The list of symbolic bus interactions. These include both the stateless bus interactions (lookups) and the stateful bus interactions.

A circuit is satisfied under an assignment when all algebraic constraints evaluate to zero and every bus interaction message is accepted by the bus semantics:

```
/-- Whether a circuit is satisfied under a given assignment and bus semantics,
    i.e., whether it satisfies all algebraic constraints and every active bus
    interaction message is accepted. -/
def Circuit.satisfies (circuit : Circuit p) (busSemantics : BusSemantics p)
    (assignment : Variable → ZMod p) : Prop :=
  (∀ c ∈ circuit.algebraicConstraints, c.eval assignment = 0) ∧
  (∀ bi ∈ circuit.busInteractions,
    let message := bi.eval assignment
    message.multiplicity ≠ 0 → busSemantics.accepts message)
```

6 Bus Semantics

Bus semantics capture the *zkVM-specific* semantics of the buses.

structure

```
BusSemantics (p : ℕ) : Type
```

The bus semantics of the zkVM.

Constructor

`BusSemantics.mk`

Fields

`isStateful : ℕ → Bool`

Whether the bus of the given ID changes the state of the VM. Stateless bus interactions are typically lookups.

`accepts : BusInteraction (ZMod p) → Prop`

Whether the receiving chip accepts this bus interaction message, i.e. sending it violates no constraint in *another* chip. A message that is *not* accepted is, for example, one contradicting a lookup table entry. Only consulted for messages with nonzero multiplicity.

`maintainsInvariants : BusInteraction (ZMod p) → Prop`

Whether sending this bus interaction message maintains the invariants on which soundness of the system depends. For example, a memory bus might have the invariant that all sent values must be in a certain range. Only consulted for messages with nonzero multiplicity.

`admissible : List (BusInteraction (ZMod p)) → Prop`

A property on *stateful* bus messages with nonzero multiplicity. Completeness is only required for assignments whose stateful messages are `admissible`. One useful way to use this is to describe the semantics of memory buses, see `ApcOptimizer/MemoryBus.lean`.

Instances exist for OpenVM and SP1. For example, the OpenVM bus semantics defines the following:

- `isStateful`: The ”execution bridge” and ”memory” buses are stateful, while all other buses are stateless.
- `accepts`: For all lookups, this checks whether the given bus message is in the precomputed table. Also, since OpenVM’s circuits maintain the invariant that only range-checked values are sent to the register and memory address spaces, a memory bus *receive* checks that the received value is in the correct range.
- `maintainsInvariants`: Checks whether all multiplicities are 1 for stateless buses, and 1 or -1 for stateful buses. Also, for memory bus interactions, it checks that the values are in the correct range. This also applies to *sent* values, ensuring that the optimized circuit does not violate the invariant mentioned above.
- `admissible`: All symbolic bus interactions to stateful buses are ordered by *time*. Also, we assume that register `x0` always returns `0`.

7 Soundness

Soundness is arguably the most important property that must be guaranteed by the optimizer. Intuitively, it states that anything the optimized circuit accepts, the original would have accepted too, with the same effect on the rest of the system.

First, we define the side effects of a circuit under an assignment as the net effect it has on the stateful buses.

```
/-- The side effects of a circuit under a given assignment and bus semantics:
    the net multiplicity with which each tuple is sent to a *stateful* bus. -/
def Circuit.sideEffects (circuit : Circuit p) (busSemantics : BusSemantics p)
    (assignment : Variable → ZMod p) : BusState p :=
  fun message =>
    ((circuit.busInteractions.map (fun bi => bi.eval assignment)).filter
```

```
      (fun m => busSemantics.isStateful m.busId &&
        decide ((m.busId, m.payload) = message))).map
      (fun m => m.multiplicity) |>.sum
```

Second, we define that a circuit *guarantees invariants* if, under any satisfying assignment, every bus interaction maintains the invariants of the bus semantics.

```
/-- Whether a circuit guarantees that all invariants are maintained under a
    given bus semantics. -/
def Circuit.guaranteesInvariants (circuit : Circuit p)
    (busSemantics : BusSemantics p) : Prop :=
  ∀ assignment, circuit.satisfies busSemantics assignment →
    ∀ bi ∈ circuit.busInteractions,
      let message := bi.eval assignment
      message.multiplicity ≠ 0 → busSemantics.maintainsInvariants message
```

Finally, we formalize what it means for an optimized circuit to be a sound replacement for an original circuit:

```
/-- Whether an optimized circuit is a sound replacement for an original
    circuit. Informally, for any satisfying assignment of the optimized
    circuit, there exists a corresponding satisfying assignment of the original
    circuit *with equal side effects*. Also, the optimized circuit must
    maintain all invariants guaranteed by the original circuit. -/
def Circuit.isSoundReplacementOf (optimizedCircuit originalCircuit : Circuit p)
    (busSemantics : BusSemantics p) : Prop :=
  (∀ assignment, optimizedCircuit.satisfies busSemantics assignment →
    ∃ assignment', originalCircuit.satisfies busSemantics assignment' ∧
      optimizedCircuit.sideEffects busSemantics assignment =
        originalCircuit.sideEffects busSemantics assignment') ∧
  (originalCircuit.guaranteesInvariants busSemantics →
    optimizedCircuit.guaranteesInvariants busSemantics)
```

8 Completeness

The *completeness* property ensures that for any valid zkVM execution, the prover *can* construct a satisfying assignment for the optimized circuit.

8.1 Admissible assignments

First, we define what it means for an assignment to be *admissible* under a bus semantics. An assignment is admissible if all active stateful messages satisfy the bus semantics' `admissible` predicate:

```
/-- Whether a given assignment is admissible under the bus semantics. -/
def Circuit.admissible (circuit : Circuit p) (busSemantics : BusSemantics p)
```

```
    (assignment : Variable → ZMod p) : Prop :=
  busSemantics.admissible
    ((circuit.busInteractions.map (fun bi => bi.eval assignment)).filter
      (fun m => decide (m.multiplicity ≠ 0) && busSemantics.isStateful m.busId))
```

Completeness is only required for admissible assignments.

8.2 Witness generation

Second, we need to guarantee that the prover can also compute a satisfying assignment for the optimized circuit. To this end, the optimizer emits a list of *derivations*, specified in a custom witness-generation IR:

inductive type

```
ComputationMethod (p : ℕ) : Type
```

A method for computing a *derived* variable's value from other variables, mirroring powdr's `ComputationMethod`. For newly introduced variables, this is interpreted by powdr's witness generator.

Constructors

| `ComputationMethod.const {p : ℕ} (c : ZMod p) : ComputationMethod p`

A constant value.

| `ComputationMethod.quotientOrZero {p : ℕ} (num den : Expression p) : ComputationMethod p`

The quotient of two expressions, or zero if the denominator is zero.

| `ComputationMethod.ifEqZero {p : ℕ} (cond : Expression p) (thenM elseM : ComputationMethod p) : ComputationMethod p`

Conditional computation: if `cond` evaluates to zero, use `thenM`, else use `elseM`.

```
/-- A list of derived variables paired with how to compute each, consumed by
    witness generation. -/
abbrev Derivations (p : ℕ) := List (Variable × ComputationMethod p)
```

With the data structures in place, we can define a prescribed witness generation algorithm that we expect the prover to implement. The algorithm derives a valid assignment for the optimized circuit from a valid assignment for the input circuit. In essence, for each variable in the output circuit:

- If it is a powdr-ID variable, it is reused from the input assignment.
- If it is a derived variable, the optimizer must have emitted a computation method for it. The witness generation algorithm evaluates this method under the input assignment to compute the output variable's value.

```
/-- The `ComputationMethod` witness generation uses for `v`. If `v` appears
    multiple times, the last derivation is returned; `none` if `v` has no
```

```
    derivation. -/
def Derivations.methodFor :
    Derivations p → Variable → Option (ComputationMethod p)
  | [], _ => none
  | (u, cm) :: rest, v =>
      match Derivations.methodFor rest v with
      -- If `v` is derived later, that derivation overrides this one.
      | some later => some later
      | none => if u = v then some cm else none

/-- Whether `ds` lets witness generation produce every element of `outputVars`
    from `inputVars`: each output variable is either an input variable (reused)
    or a derived variable with a method that reads only input variables. -/
def Derivations.cover (ds : Derivations p)
    (inputVars outputVars : List Variable) : Prop :=
  ∀ v ∈ outputVars,
    match v.powdrId? with
    | some _ => v ∈ inputVars
    | none => ∃ cm, ds.methodFor v = some cm ∧ ∀ x ∈ cm.vars, x ∈ inputVars

/-- Witness generation: reconstruct an output assignment from an input
    assignment. Every powdr-ID (input) variable passes through unchanged; every
    other variable is computed by the method `ds` records for it, read from the
    input variables. This is what powdr runs to fill the optimized circuit's
    variables from an input trace. -/
def Derivations.witgen (ds : Derivations p)
    (inputAssignment : Variable → ZMod p) (v: Variable) : ZMod p :=
  match v.powdrId? with
  -- Note that by `Derivations.cover`, if `v` appears in the output circuit,
  -- it must also exist in the input circuit, so this case is always
  -- well-defined.
  | some _ => inputAssignment v
  | none =>
    match Derivations.methodFor ds v with
    | some cm => cm.eval inputAssignment
    -- Note that by `Derivations.cover`, if `v` appears in the output
    -- circuit, this case is impossible.
    | none => inputAssignment v
```

8.3 The full completeness property

Putting the pieces together, we define what it means for an optimized circuit to be a *complete* replacement for an original circuit. Structurally, the returned derivations must contain no unused entries and must cover every output variable from the input variables. Semantically, every admissible satisfying input assignment must produce a satisfying and admissible output assignment with equal side effects.

```
/-- Whether an optimized circuit is a complete replacement for an original
↪  circuit. -/
def Circuit.isCompleteReplacementOf
    (optimizedCircuit originalCircuit : Circuit p)
    (busSemantics : BusSemantics p) (ds : Derivations p) : Prop :=

  -- ASSUMPTION: every variable in the original circuit has a powdr ID.
  (∀ v ∈ originalCircuit.vars, v.powdrId?.isSome) →

  -- `ds` does not contain unused derivations.
```

```
  (∀ derivation ∈ ds, derivation.1 ∈ optimizedCircuit.vars) ∧

  -- The optimized circuit variables can be derived from the original circuit
  -- variables, and the return derivations.
  ds.cover originalCircuit.vars optimizedCircuit.vars ∧

  -- For any admissible satisfying assignment of the original circuit, the
  -- optimized circuit is also satisfied and admissible, with equal side
  -- effects, under the assignment produced by witness generation.
  ∀ assignment,
    originalCircuit.admissible busSemantics assignment →
    originalCircuit.satisfies busSemantics assignment →
    let assignment' := Derivations.witgen ds assignment
    optimizedCircuit.satisfies busSemantics assignment' ∧
      optimizedCircuit.admissible busSemantics assignment' ∧
      originalCircuit.sideEffects busSemantics assignment =
        optimizedCircuit.sideEffects busSemantics assignment'
```

9 Degree bound

The *multiplicative degree* of an expression is defined structurally: constants have degree 0, variables have degree 1, addition takes the maximum, and multiplication adds the degrees.

```
/-- The multiplicative degree of an expression. -/
def Expression.degree : Expression p → Nat
  | .const _ => 0
  | .var _ => 1
  | .add e1 e2 => max e1.degree e2.degree
  | .mul e1 e2 => e1.degree + e2.degree
```

A *degree bound* specifies the maximum multiplicative degrees allowed for algebraic constraints and bus interactions. The proving backend enforces these bounds, so the optimizer must respect them: a within-bound input yields a within-bound output.

structure

```
DegreeBound : Type
```

Bounds on the multiplicative degree of a circuit's expressions.

Constructor

```
DegreeBound.mk
```

Fields

```
identities : ℕ
```

The maximum multiplicative degree of the algebraic constraints.

```
busInteractions : ℕ
```

The maximum multiplicative degree of the bus interactions.

Given a zkVM-specific degree bound and an optimizer, we state what it means for the optimizer to respect the bound: For any input circuit that is within the bound, the output circuit must also be within the bound.

```
/-- Whether a circuit stays within a degree bound. -/
def Circuit.withinDegree (circuit : Circuit p) (b : DegreeBound) : Prop :=
  (∀ c ∈ circuit.algebraicConstraints, c.degree ≤ b.identities) ∧
  (∀ bi ∈ circuit.busInteractions,
    bi.multiplicity.degree ≤ b.busInteractions ∧
      ∀ e ∈ bi.payload, e.degree ≤ b.busInteractions)

/-- Whether an optimizer respects a degree bound: a within-bound input always
    yields a within-bound output. -/
def optimizerRespectsDegreeBound (b : DegreeBound)
    (optimizer : Circuit p → Circuit p × Derivations p) : Prop :=
  ∀ circuit : Circuit p,
    circuit.withinDegree b →
    (optimizer circuit).1.withinDegree b
```

10 Optimizer

Putting the pieces together, we define what it means for an optimizer to be *correct*. An *optimizer* is a function that maps a circuit to a new circuit and a list of derivations.

```
abbrev Optimizer (p : ℕ) := Circuit p → Circuit p × Derivations p
```

An optimizer is correct if, for every input circuit, replacing it with the optimized circuit is both sound and complete, and the optimizer respects the degree bound.

```
/-- An optimizer is correct if, for every input circuit, replacing it with the
    optimized circuit is both sound and complete, and the optimizer respects
    the degree bound `b`. -/
def Optimizer.isCorrect (optimizer : Optimizer p)
    (busSemantics : BusSemantics p) (b : DegreeBound) : Prop :=
  (∀ originalCircuit : Circuit p,
    let (optimizedCircuit, derivations) := optimizer originalCircuit
    (optimizedCircuit.isSoundReplacementOf originalCircuit busSemantics) ∧
    (optimizedCircuit.isCompleteReplacementOf originalCircuit busSemantics
      derivations))
  ∧ optimizerRespectsDegreeBound b optimizer
```