A Fistful of Abstract Values: Crafting the Transformers Compilers Actually Need

DOMINIC KENNEDY[†], University of Utah, USA
XUANYU PENG[†], University of California San Diego, USA
YUYOU FAN, University of Utah, USA
JOHN REGEHR, University of Utah, USA
LORIS D'ANTONI, University of California San Diego, USA
BEN GREENMAN, University of Utah, USA

Precise dataflow analyses are essential to modern compiler optimization, but their effectiveness depends on carefully engineered abstract transformers for the instruction idioms that occur in real programs. In LLVM, these transformers have been refined manually over many years, typically in response to human observations about missed facts and lost optimization opportunities.

This paper introduces `SimplyTheBest`, a data-driven technique for improving dataflow analyses automatically. Rather than requiring compiler developers to notice and hand-code each missing case, `SimplyTheBest` mines recurring instruction patterns where a current analysis loses precision and where a more precise interpretation could expose new optimizations. For each such pattern, `SimplyTheBest` uses a formal specification of the instructions being analyzed to construct a maximally precise abstract transformer for the specific abstract input values that are seen during compilation. Our key observations are: (1) a large corpus of software is available to inform offline synthesis, (2) analyses in LLVM exercise only a small fraction of their abstract domains, and (3) aggressive minimization of that small fraction makes it practical to store the resulting abstract values as fast lookup tables.

On the high-profile dataset `llvm-opt-benchmark`, `SimplyTheBest` discovers millions of known bits beyond LLVM's hand-engineered analysis and enables optimizations such as removing a load and replacing a function by a constant. These results show that compiler analyses can be systematically improved by mining the precision gaps that matter and synthesizing inexpensive transformers to close them.

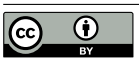
CCS Concepts: • **Software and its engineering → Compilers**; *Automatic programming*; • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: Abstract Interpretation, SMT, LLVM

ACM Reference Format:
Dominic Kennedy, Xuanyu Peng, Yuyou Fan, John Regehr, Loris D'Antoni, and Ben Greenman. 2027. A Fistful of Abstract Values: Crafting the Transformers Compilers Actually Need. *Proc. ACM Program. Lang.* 11, POPL, Article 32 (January 2027), 33 pages. https://doi.org/10.1145/3871734

[†]Equal contribution

Authors' Contact Information: Dominic Kennedy, University of Utah, USA, dominicmkennedy@gmail.com; Xuanyu Peng, University of California San Diego, USA, xup002@ucsd.edu; Yuyou Fan, University of Utah, USA, yuyou.fan@utah.edu; John Regehr, University of Utah, USA, regehr@cs.utah.edu; Loris D'Antoni, University of California San Diego, USA, ldantoni@ucsd.edu; Ben Greenman, University of Utah, USA, benjamin.l.greenman@gmail.com.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2027 Copyright held by the owner/author(s).
ACM 2475-1421/2027/1-ART32
https://doi.org/10.1145/3871734

1 Introduction

Modern compiler infrastructures such as LLVM and MLIR use lightweight dataflow analyses such as known bits (individual bits are either always zero, always one, or unknown) and integer ranges to discover facts that are true across all executions of a program. These facts then drive optimizations. All of these analyses are instances of abstract interpretation [6]: each instruction is assigned an abstract transformer that maps facts about its inputs to facts about its output. Unlike heavyweight analyzers such as Astrée [9], Polyspace [34], and Frama-C [24], which use sophisticated domains and path/context sensitivity to prove system-level safety properties, compiler analyses must be cheap enough to run routinely, even in settings such as just-in-time compilation in QEMU [1], which employs a quirky but recognizable known bits analysis, and HotSpot, which uses both known bits and integer ranges in its higher-tier compiler [17]. The precision of such an analyzer depends not only on the abstract domain(s) in use, but also on how well their transformers capture the actual idioms that occur in real programs.

In LLVM and MLIR, many of these transformers have been refined by hand over years of compiler development. This process has produced highly effective analyses, but it is fundamentally ad hoc: developers must notice a missed fact, identify a general root cause, and implement a specialized transformer. As a result, important cases are easy to miss, especially when precision depends on constants, repeated uses of the same SSA value, or reasoning about a collection of instructions together. When new instructions or intrinsics are added to these compilers, it can take years before dataflow analyses are extended to support them with high-quality abstract transformers.

This paper introduces `SimplyTheBest`, an automated technique for transformer specialization. `SimplyTheBest` identifies the instruction patterns where additional precision is likely to matter, synthesizes specialized transformers for those patterns, and deploys them as simple lookup tables that are cheap enough to use inside existing compiler analyses. On multi-million-line projects such as CPython and V8, analysis and specialization run overnight, well under twelve hours.

1.1 Transformer Specialization Today

The need for specialization is easy to see in a simple idiom. The expression `x & (x-1)` clears the lowest set bit of `x` and is seen 2,315,037 times by LLVM's known bits analysis when analyzing our input corpus, `llvm-opt-benchmark` [54]. A transformer that analyzes this whole pattern can infer that the least significant output bit is always `0` because `x` and `x-1` have opposite parity, so their conjunction must be even. However, a dataflow analysis that applies only general-purpose transformers instruction-by-instruction loses this relationship: if `x` is completely unknown, then so is `x-1`, and the final `and` operation produces no known bits. The missed fact is shallow, cheap, and useful, but it requires recognizing the instruction sequence as a pattern rather than analyzing each instruction in isolation. LLVM special-cases this pattern, as well as other, more domain-specific ones. For example, its `computeKnownBitsFromLerpPattern()` matches linear-interpolation code such as `a * (b - c) + c * d` that is common in graphics applications.

The current version of LLVM contains numerous recognizers for specific idioms, all implemented in an ad hoc fashion. Some of these transformers gain precision by exploiting the fact that one operand is a literal constant; others exploit repeated uses of the same SSA value; others exploit semantic case splits, such as parity or sign; and still others are composite, gaining precision by analyzing a small collection of instructions as a unit. These specializations can be surprisingly important: prior work on bulk transformer synthesis showed that composite transformers for idioms such as `abs`, `popcount`, `min`, and `max` can be dramatically more precise than directly composing sequential transformers [42]. The challenge that we address is systematically discovering which specializations are worth synthesizing, and then deploying them efficiently in production analyses.

1.2 From Manual Specialization to Profile-Guided Synthesis

`SimplyTheBest` turns transformer specialization into a profile-guided synthesis problem. Given a corpus of compiler IR, `SimplyTheBest` mines recurring instruction patterns that exhibit the ingredients of useful specialization, such as repeated SSA values and multi-instruction structure. It then evaluates these candidates by measuring their precision gap: the difference between what existing transformers infer and what an optimal transformer could infer. `SimplyTheBest` uses a set of completeness criteria for known bits and integer ranges to identify when existing transformers are already as precise as possible, allowing `SimplyTheBest` to efficiently rule out patterns with no precision left to recover. For the remaining patterns, `SimplyTheBest` uses an SMT-based synthesis infrastructure, built on LLVM's semantics and existing abstract domains, to compute optimal abstract outputs and generate specialized transformers.

The key discovery that makes `SimplyTheBest` practical is an empirical sparsity property of compiler analyses:

> *Real compiler analyses exercise only a tiny fraction of their abstract domains.*

In principle, a precise transformer for an instruction sequence may have an enormous input space, which would require writing or synthesizing a complex abstract transformer that can handle all such cases. We explored this kind of synthesis in the past [42], but found it to be ineffective on sequences because, in practice, most instruction patterns are encountered on only a small set of abstract values. `SimplyTheBest` exploits this sparsity by representing each synthesized transformer as a lookup table from relevant abstract inputs to precise abstract outputs, where each output is computed offline using an SMT solver. The tables are then minimized and embedded into LLVM's analysis, yielding transformers that are simple, precise, and cheap to apply during compilation.

Contributions. In summary, this paper makes the following contributions:

- We present `SimplyTheBest`, a profile-guided framework for improving compiler dataflow analyses by mining recurring instruction patterns, identifying recoverable precision gaps, and synthesizing specialized transformers (overview in Section 2).
- We synthesize specialized transformers as compact lookup tables, exploiting sparsity in observed abstract values to make the transformers practical (Section 3).
- We develop a mining procedure that collects recurring LLVM IR patterns and their observed abstract values, using completeness criteria to rule out patterns where specialization provably cannot improve precision (Section 4). The completeness analysis covers 63 LLVM IR operations and is formalized in Lean (Section 4.2).
- We implement `SimplyTheBest` (Section 5) for LLVM's `KnownBits` and `ConstantRange` domains and evaluate it on `llvm-opt-benchmark`, a collection of 240 high-profile LLVM projects, showing that it discovers 37 million and 15 million additional known bits on each domain respectively, with a median compilation-time overhead of 0.25–1.5 % (Section 6). All of this additional precision comes from compact lookup tables that cumulatively store fewer than 50,000 rows of abstract values.

`SimplyTheBest` creates efficient, precise, and correct abstract transformers using push-button methods. The results show that profile-guided synthesis can recover precision left behind by years of manual specialization, while producing transformers that are simple enough to deploy.

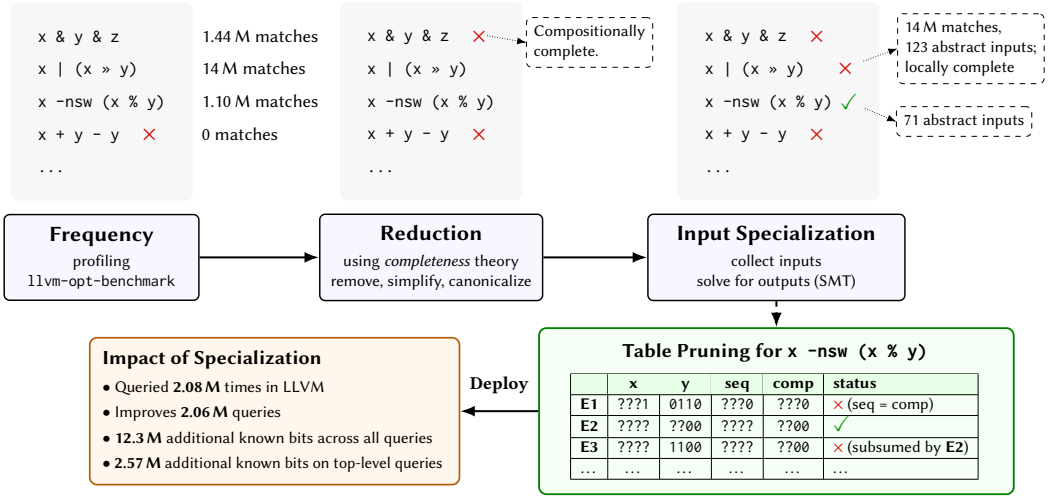


Fig. 1. SimplyTheBest mines frequent patterns from llvm-opt-benchmark, reduces compositionally complete subterms in patterns, and finally uses profiling and SMT to specialize the remaining patterns for the abstract inputs observed in practice. Rows are kept aligned across stages for comparison; × marks rejected patterns and pruned items while ✓ marks accepted patterns and selected items.

2 Overview of the Approach

We illustrate the main ideas of SimplyTheBest by showing how it discovers the pattern `x -nsw (x%y)` and synthesizes a specialized abstract transformer for it in LLVM's known bits analysis (Figure 1). Known bits is a lightweight integer abstraction that records, for each bit of a value, whether that bit is definitely `0`, definitely 1, or unknown. For example, `??00` denotes an integer whose two least significant bits are known to be zero, while `????` denotes an integer about which no bits are known. In this example, `-nsw` denotes subtraction with LLVM's *no-signed-wrap* flag, which lets the analysis reason under the assumption that the subtraction does not signed-overflow.[1]

In the following, we call the existing instruction-by-instruction analysis a *sequential* transformer: it first abstracts the result of `x%y` and then analyzes the subtraction. By contrast, a *composite* transformer analyzes the whole pattern `x -nsw (x%y)` at once, avoiding the intermediate abstraction that can lose the relationship between `x` and `x%y`. This pattern could be profitable to specialize because it exposes a useful fact that sequential reasoning loses: whenever `y` has k known trailing zeros, so does `x -nsw (x%y)`. Indeed, if `y` is a multiple of 2^k, then `x -nsw (x%y)` is also a multiple of 2^k, because `x%y` has the same lowest k bits as `x`. Thus the lowest k bits of the result are zero.

Does the pattern appear often? A pattern is worth specializing only if LLVM's analysis actually encounters it. To find such patterns, SimplyTheBest profiles LLVM's existing demand-driven analysis. Each call to `ComputeKnownBits()` asks for known bits about one SSA value and recursively queries the values on which it depends. SimplyTheBest records the instruction patterns exposed by these queries and counts how often they occur. Across llvm-opt-benchmark, the pattern `x -nsw (x%y)` matches 1.10 million requests for known bits information issued by the compiler. In contrast, the pattern `x + y - y`, a classic pattern for which sequential reasoning forgets that the two occurrences of `y` are the same value, occurs zero times. This is because LLVM has an "early

[1]When overflow occurs, the result is not immediate undefined behavior, but rather a *poison value*. LLVM's static analyses are designed to be poison-oblivious: they reason only about the non-poison case.

common subexpression elimination" pass that simplifies the pattern away before the compiler invokes known bits.

Would specializing the pattern help improve precision? Frequent occurrence alone, however, is not enough. A pattern can be frequent but already handled optimally by sequential reasoning. The pattern `x & y & z` matches 1.44 million requests, but its sequential transformer is already optimal for the known bits domain—i.e., a composite transformer cannot improve on the sequential one. We use the notion of *compositional completeness* to identify such cases [11].

Even when a sequential transformer is not optimal on all abstract values, it can be optimal for the abstract values the analysis encounters in practice. For example, `x | (x » y)` occurs 14 million times and exercises 123 distinct abstract inputs, but on every observed input the bits of the shift amount `y` are all known. In this regime, the sequential and composite transformers coincide, so the pattern is locally complete on the inputs that LLVM actually observes.

A fistful of abstract values. Our running example, the pattern `x -nsw (x%y)`, passes these filters. It is frequent, and it is not compositionally complete in the known bits domain: applying the sequential transformer on some of the encountered abstract inputs can produce an abstract output strictly less precise than the optimal composite transformer for the whole pattern. Thus, the pattern is both common enough to matter and capable of recovering precision that LLVM's existing per-instruction transformers lose.

The next question is whether a useful transformer can be made cheap enough to run during compilation. In principle, the space of abstract inputs is enormous. For example, a two-instruction pattern `op1(x, op2(y, z))` over 64-bit known bits values has an input space of $3^{3 \cdot 64}$, since each bit of each input may be zero, one, or unknown.

As hinted above and in our introduction, LLVM asks far fewer distinct questions than it might. Although `x -nsw (x%y)` is invoked 1.10 million times across `llvm-opt-benchmark`, and despite an input space of size $3^{2 \cdot 64}$ (it has only two arguments), these invocations exercise only *71 distinct abstract inputs*—**just a fistful of abstract values**! For most of these inputs, the optimal composite transformer produces a more precise output than the optimal sequential transformer, confirming that the pattern has precision gaps on the workload LLVM actually presents to the analysis.

This is the point where `SimplyTheBest` purposefully departs from prior work on general-purpose transformer synthesis [19, 42]. It does not attempt to synthesize a closed-form transformer over the entire abstract domain, nor is it meant to replace complex domain-specific reasoning in expressive analyzers. Instead, `SimplyTheBest` targets the regime that production compilers rely on heavily: lightweight abstract domains whose queries are numerous, repetitive, and sparse.

Building a compact transformer. For each of the 71 observed abstract inputs, `SimplyTheBest` reuses existing SMT-based techniques for computing maximally precise abstract transformer outputs [49]. The infrastructure encodes the concrete semantics of the instruction pattern together with the constraints imposed by the input abstract values, then asks the solver which output facts are sound for all concrete executions represented by those inputs. `SimplyTheBest` uses this procedure to compute two outputs: the best result obtainable by sequentially composing the existing per-instruction transformers, and the best result obtainable by analyzing the whole pattern as one concrete function. Inputs on which these two outputs coincide are dropped, since specialization cannot improve precision there. The remaining inputs become candidate entries in a lookup table mapping abstract inputs to optimal composite outputs.

Figure 1 illustrates how the table is built and its entries are efficiently pruned for the pattern `x -nsw (x%y)`. Suppose profiling observes three candidate table entries. In entry **E1**, the input is `x = ???1` and `y = 0110`; both the sequential transformer and the composite transformer return

`???0`. Since the specialized transformer adds no precision on this input, `SimplyTheBest` removes the entry. In entry **E2**, the input is `x = ????` and `y = ??00`; the sequential transformer returns `????`, but the composite transformer returns `??00`, so the entry is useful and kept. Entry **E3** has input `x = ????` and `y = 1100`, and its composite output is also `??00`. However, **E3** is redundant because it is covered by **E2**: every value represented by `1100` is also represented by `??00`, so **E2**'s more general input already applies and produces the same output. After applying this pruning to all observed inputs, `SimplyTheBest` represents the transformer for this pattern as a lookup table with only 40 entries.

The deployed transformer is still more general than a plain key-value cache. Given a query input, `SimplyTheBest` finds every retained table entry whose input covers the query input and returns the meet of their composite outputs. This is sound because each covering entry was synthesized for a more general abstract input, and therefore applies to every more-precise input it covers. Taking the meet lets multiple applicable entries contribute precision while preserving soundness.

Deploying the transformer. When deployed inside LLVM, the 40-entry table for `x -nsw (x%y)` is queried 2.08 million times and improves precision on 2.06 million of those queries. The number of deployed queries is larger than the 1.10 million invocations observed during profiling because greater precision allows LLVM's recursive analysis to continue farther along backward SSA edges, exposing additional uses of the same pattern. Across `llvm-opt-benchmark`, this single composite transformer produces 12.3 million additional known bits across all queries to `ComputeKnownBits()`, or 2.57 million additional known bits when counting only top-level queries. Thus, for this pattern, **a fistful of abstract values—71 observed inputs, pruned to 40 retained entries—is enough to recover millions of missing facts during compilation!**

3 Synthesizing Best Local Abstract Transformers

This section gives the semantic foundation for `SimplyTheBest`'s synthesized transformers. We first define the abstract domains used throughout the paper and recall the standard notion of a best abstract transformer. We then adapt this notion to the profile-guided setting: instead of requiring a transformer to be optimal on the entire abstract domain, `SimplyTheBest` synthesizes a sound transformer that is optimal on the abstract inputs observed during profiling. Finally, we show how these best local transformers for instruction patterns are computed with SMT and are represented as compressed lookup tables that augment a baseline analysis.

3.1 Abstract Domains

An abstract domain $\mathcal{A}$ is a partially ordered set of abstract values, where each $A \in \mathcal{A}$ describes a set of concrete values from a concrete domain C. We write $A_1 \sqsubseteq A_2$ when A_1 is at least as precise as A_2, meaning that A_1 describes a subset of the concrete values described by A_2. The bottom element $\bot \in \mathcal{A}$ describes the empty set, and the top element $\top \in \mathcal{A}$ describes all concrete values.

In this paper, the concrete domain is LLVM's integer type `iN`: bitvectors of width `N`. We instantiate the framework with three lightweight integer domains: `KnownBits`, unsigned integer ranges (`UConstRange`), and signed integer ranges (`SConstRange`). These domains correspond to the value analyses that production compilers can typically afford to run during optimization. In general, `SimplyTheBest` applies to any domain where the abstract values form a lattice and are related to the concrete domain via a Galois connection (detailed below).

We use two range domains instead of the one domain from LLVM (`ConstantRange`) because the elements of LLVM's domain do not form a lattice. In `ConstantRange`, intervals may wrap around both unsigned and signed boundaries. At bitwidth 8, for example, the range $[255, 2)$ represents the set $\{255, 0, 1\}$. This representation is useful in LLVM because clients may want ranges that respect

either the signed or unsigned interpretation of the given variable. But, `SimplyTheBest` needs lattice structure in order to compute maximally precise abstract outputs. Our domain `UConstRange` contains unsigned intervals that do not wrap over the unsigned boundary, and `SConstRange` contains signed intervals that do not wrap over the signed boundary. The set $\{255, 0, 1\}$ is represented as $\top$ by `UConstRange`, and as the signed interval $[-1, 1]$ in `SConstRange`, since the bitvector 255 represents -1 when read as a signed integer.

We relate concrete and abstract values using a Galois connection $\mathcal{P}(C) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$, where the *concretization* function γ maps an abstract value to the set of concrete values it represents, and the *abstraction* function α maps a set of concrete values to the most precise abstract value that represents it. Equivalently, for every concrete set $C \subseteq C$ and abstract value $A \in \mathcal{A}$, we have that $\alpha(C) \sqsubseteq A$ iff $C \subseteq \gamma(A)$. This relationship captures both *soundness* and *precision*: $\alpha(C)$ overapproximates C, and is at least as precise as any other abstract value that does so.

3.2 Best Local Transformers

An immediate consequence of a Galois connection is a canonical notion of optimal precision. For a concrete operation $f : C^k \to C$, let $\widetilde{f} : \mathcal{P}(C)^k \to \mathcal{P}(C)$ be its collecting semantics: $\widetilde{f}(C_1, \ldots, C_k) = \{ f(c_1, \ldots, c_k) \mid c_i \in C_i \text{ for each } i \}$.

Definition 3.1 (Best Abstract Transformer [6]). Let $f : C^k \to C$ be a concrete operation. The *best abstract transformer* for f maps abstract inputs to the most precise abstract value containing all possible concrete outputs, and is defined as follows:

$$f^\sharp_{\text{best}}(A_1, \ldots, A_k) = \alpha\big(\widetilde{f}(\gamma(A_1), \ldots, \gamma(A_k))\big).$$

By construction, $f^\sharp_{\text{best}}$ is sound. Moreover, for any sound abstract transformer $g^\sharp$ for f and any abstract inputs $A_1, \ldots, A_k$, we have $f^\sharp_{\text{best}}(A_1, \ldots, A_k) \sqsubseteq g^\sharp(A_1, \ldots, A_k)$. Thus every other sound transformer is no more precise than $f^\sharp_{\text{best}}$ on every input.

For example, consider the pattern `x -nsw (x % y)` from Section 2. On the abstract input $x =$ `????` and $y =$ `??00`, every concrete value represented by y is a multiple of 4. Therefore `x % y` has the same two low bits as `x`, and subtracting it from `x` produces a result whose two low bits are zero. The best abstract transformer for the whole pattern therefore returns `??00` on this input. On the same input, the sequential transformer is less precise, returning `????`.

The definition of the best transformer specifies a goal, but it does not by itself give a practical representation of the transformers. Automatically generating efficient best transformers is a difficult problem; prior work aims to produce sound, maximally precise, and general-purpose transformers for whole abstract domains [19, 42, 44]. `SimplyTheBest` instead exploits empirical sparsity: rather than synthesizing one $f^\sharp_{\text{best}}$ to handle every abstract input, it focuses on the subset $\mathcal{S}$ of abstract inputs observed during profiling (we discuss in Section 4.3 how $\mathcal{S}$ is collected in practice).

Definition 3.2 (Best Local Abstract Transformer). Let $f : C^k \to C$ be a concrete operation, and let $\mathcal{S} \subseteq \mathcal{A}^k$ be a set of abstract inputs. We say that a transformer $f^\sharp : \mathcal{A}^k \to \mathcal{A}$ is **a** *best local abstract transformer* for f on $\mathcal{S}$ if it is sound everywhere,

$$\forall (A_1, \ldots, A_k) \in \mathcal{A}^k. \quad f^\sharp_{\text{best}}(A_1, \ldots, A_k) \sqsubseteq f^\sharp(A_1, \ldots, A_k),$$

and maximally precise on $\mathcal{S}$,

$$\forall (A_1, \ldots, A_k) \in \mathcal{S}. \quad f^\sharp_{\text{best}}(A_1, \ldots, A_k) = f^\sharp(A_1, \ldots, A_k).$$

Definition 3.2 requires optimality only on inputs in $\mathcal{S}$, so there can be many best local abstract transformers. They may differ on inputs outside of $\mathcal{S}$, but they are all sound everywhere and equivalent on $\mathcal{S}$.

For example, suppose profiling observes only the abstract input $\mathcal{S} = \{(\texttt{????}, \texttt{??00})\}$ for the pattern $pat(x, y) = \texttt{x -nsw (x \% y)}$. A best local abstract transformer for pat on $\mathcal{S}$ must return the optimal output `??00` on (`????`, `??00`), and it must remain sound on every other abstract input. However, it need not be optimal outside $\mathcal{S}$; for example, returning $\top$ for all other inputs is allowed.

3.3 Synthesizing Best Local Transformers as Compressed Lookup Tables

To synthesize a best local transformer $f^\sharp$ for a function f on profiled inputs $\mathcal{S}$, `SimplyTheBest` must compute $f^\sharp_{\text{best}}(A_1, \ldots, A_k)$ for each $(A_1, \ldots, A_k) \in \mathcal{S}$. Definition 3.1 gives us a very inefficient way to perform this computation: concretize all inputs in $(A_1, \ldots, A_k)$, run f on all such inputs, and then abstract. This approach is intractable when the inputs are large bitvectors.

3.3.1 Synthesizing Best Abstract Outputs using SMT Solvers. `SimplyTheBest` takes advantage of the fact that a first-order encoding of the semantics of f is readily available, and thus opts for an SMT-friendly characterization of the problem: the best abstract output is exactly the most precise sound output.

Proposition 3.3 (Maximally Precise Output Computation). *Let $f : C^k \to C$ and $\vec{A} \in \mathcal{A}^k$. An abstract value $O \in \mathcal{A}$ is $f^\sharp_{\text{best}}(\vec{A})$ iff it is sound, and every more-precise abstract value is unsound:*

$$O = f^\sharp_{\text{best}}(\vec{A}) \iff \widetilde{f}(\gamma(\vec{A})) \subseteq \gamma(O) \ \wedge \ \big(\forall O' \in \mathcal{A}.\ O' \sqsubset O \implies \widetilde{f}(\gamma(\vec{A})) \not\subseteq \gamma(O')\big).$$

Before showing how this check can be encoded using SMT solvers, we note that solving the quantified formula in Proposition 3.3 is expensive: the universal quantifier encodes the requirement that every more-precise output be unsound.

Rather than solving this quantified formula directly, `SimplyTheBest` constructs the output by asking a sequence of quantifier-free soundness queries. For `KnownBits`, each possible output fact is a candidate bit constraint: `SimplyTheBest` asks whether all concrete outputs have bit i equal to 0, and separately whether they have bit i equal to 1. Each valid fact is added to the output. For `UConstRange` and `SConstRange`, `SimplyTheBest` tightens the interval endpoints by binary search, using SMT queries to test whether a proposed bound contains all possible outputs. Thus the "descent" through the lattice is driven by the structure of the target domain. Taneja et al. [49] first applied this strategy to `KnownBits` and `ConstantRange`; we adopt it here. `SimplyTheBest` uses Bitwuzla [38] as the backend SMT solver, since it is highly optimized for bit-vector reasoning.

3.3.2 Transformers as Lookup Tables. The result of the SMT phase is a set of optimal outputs for the profiled inputs. To deploy these outputs, `SimplyTheBest` starts from an existing sound transformer for f (we call this the *base transformer*, written $f^\sharp$), and augments it with a lookup table for the values in $\mathcal{S}$. In Section 4, when f is induced by an instruction pattern, $f^\sharp$ will be the sequential transformer obtained by composing the best per-instruction transformers. The goal of the synthesized table is to contribute additional precision gained from reasoning about the pattern as a whole; when no table entry applies, the analysis falls back to $f^\sharp$.

As hinted in Section 2, our table can be used beyond specific entries that exactly match it. Specifically, any entry $\langle \vec{A} \mapsto O \rangle$ can be applied to any more-precise query input $\vec{Q} \sqsubseteq \vec{A}$ (pointwise), not only to the exact profiled input $\vec{A}$. When several entries apply, we meet their outputs. We always meet their final value with that of the base transformer as well.

Definition 3.4 (Lookup-table Transformer). Let $f : C^k \to C$ be a concrete function, and let $f^\sharp : \mathcal{A}^k \to \mathcal{A}$ be a sound abstract transformer for f. A *lookup-table transformer* for f is a pair $\mathcal{L} = (f^\sharp, T)$, where T is a finite set of entries $\langle \vec{A} \mapsto O \rangle$ with $\vec{A} \in \mathcal{A}^k$ and $O \in \mathcal{A}$.

The semantics of the transformer denoted by $\mathcal{L}$ is

$$[\![(f^\sharp, T)]\!](\vec{Q}) = f^\sharp(\vec{Q}) \sqcap (\bigsqcap \{ O \mid \langle \vec{A} \mapsto O \rangle \in T \text{ and } \vec{Q} \sqsubseteq \vec{A} \}).$$

Thus a table entry synthesized for $\vec{A}$ also applies to any more-precise input $\vec{Q}$ that it covers. For the example `x -nsw (x % y)` from Figure 1, $f^\sharp$ is the sequential transformer, and entry **E2** has input x = `????` and y = `??00`. Since `1100` $\sqsubseteq$ `??00`, entry **E2** also applies when x = `????` and y = `1100`, reusing the same synthesized output `??00`.

Lemma 3.5 (Lookup-Table Transformers are Best Local Transformers). *Given a sound abstract transformer $f^\sharp$ and a lookup table $T_0 = \{\langle \vec{A} \mapsto f^\sharp_{\text{best}}(\vec{A}) \rangle \mid \vec{A} \in \mathcal{S}\}$, the lookup-table transformer $[\![(f^\sharp, T_0)]\!]$ is a best local abstract transformer for f on $\mathcal{S}$.*

Proof. Soundness follows from monotonicity. If an entry $\langle \vec{A} \mapsto O \rangle$ covers $\vec{Q}$, then $\vec{Q} \sqsubseteq \vec{A}$, so $f^\sharp_{\text{best}}(\vec{Q}) \sqsubseteq f^\sharp_{\text{best}}(\vec{A}) \sqsubseteq O$. Thus every applicable table output is sound for $\vec{Q}$; meeting them with the sound base transformer preserves soundness.

For local optimality, let $\vec{A} \in \mathcal{S}$. The raw table contains the exact entry $\langle \vec{A} \mapsto f^\sharp_{\text{best}}(\vec{A}) \rangle$, which covers $\vec{A}$. Hence the lookup result is at least as precise as $f^\sharp_{\text{best}}(\vec{A})$. By soundness, it cannot be strictly more precise than $f^\sharp_{\text{best}}(\vec{A})$. Therefore the lookup result equals $f^\sharp_{\text{best}}(\vec{A})$. □

3.3.3 Compressing the Lookup Tables. The raw table may contain redundant entries. We define redundancy relative to the semantics of lookup-table transformers.

Definition 3.6 (Subsumed Entry). Let $(f^\sharp, T)$ be a lookup-table transformer, and let $\langle \vec{A} \mapsto O \rangle$ be an entry not necessarily in T. We say that $\langle \vec{A} \mapsto O \rangle$ is *subsumed by* $(f^\sharp, T)$ if $[\![(f^\sharp, T)]\!](\vec{A}) \sqsubseteq O$.

For raw entries where $O = f^\sharp_{\text{best}}(\vec{A})$, subsumption means that removing $\langle \vec{A} \mapsto O \rangle$ from the lookup-table will not result in losing precision on $\vec{A}$.

In practice, we encounter two common cases. First, an entry may be subsumed by the base transformer alone. For a raw entry, this means that the base transformer is already optimal on $\vec{A}$, so the table entry would only rediscover a fact the existing analysis already computes. This is the table-construction form of local completeness—i.e., $f^\sharp(\vec{A}) = f^\sharp_{\text{best}}(\vec{A}) = O$. For example, in Figure 1, entry **E1** for `x -nsw (x % y)` has input x = `???1` and y = `0110`. On this input, the sequential transformer returns `???0`, which is also the best composite output. The entry is therefore removed.

Second, an entry may be subsumed by other entries in the same table. For example, in Figure 1, entry **E3** for `x -nsw (x % y)` has input x = `????` and y = `1100` and output `??00`. Entry **E2** has input x = `????` and y = `??00` and the same output `??00`. Since `??00` is more general than `1100`, every query covered by **E3** is also covered by **E2**; and since both entries produce the same output, **E3** adds no precision.

More generally, when considering an entry $e = \langle \vec{A} \mapsto O \rangle$ in a table T, `SimplyTheBest` removes e if it is subsumed by $(f^\sharp, T \setminus \{e\})$. This transformation does not change the semantics of the transformer as long as $f^\sharp$ is monotone—i.e., if $\vec{A} \sqsubseteq \vec{B}$ component-wise, then $f^\sharp(\vec{A}) \sqsubseteq f^\sharp(\vec{B})$.

Theorem 3.7 (Entry Removal is Semantics-preserving). *Assume $f^\sharp$ is monotone. Let $e = \langle \vec{A} \mapsto O \rangle$ be an entry in a table T. If e is subsumed by $(f^\sharp, T \setminus \{e\})$, then removing e does not change the lookup-table transformer: $\forall \vec{Q}. \quad [\![(f^\sharp, T)]\!](\vec{Q}) = [\![(f^\sharp, T \setminus \{e\})]\!](\vec{Q})$.*

Proof. If e does not cover $\vec{Q}$, then e does not affect the lookup result. Otherwise, $\vec{Q} \sqsubseteq \vec{A}$. By monotonicity and subsumption, $[\![(f^\sharp, T \setminus \{e\})]\!](\vec{Q}) \sqsubseteq [\![(f^\sharp, T \setminus \{e\})]\!](\vec{A}) \sqsubseteq O$. Thus the transformer without e already produces an output at least as precise as O. □

By Theorem 3.7, `SimplyTheBest` can repeatedly and greedily remove subsumed entries without changing the transformer represented by the table. (Note that the order of removal may affect the final table, but not the transformer it denotes.) The final deployed transformer is therefore simple: given a query input, it runs the base transformer, finds the retained entries that cover the query, and meets their outputs with the base result. All SMT reasoning is performed offline; during compilation, the analysis performs only coverage tests and lattice meets.

3.4 Transformers for Instruction Patterns

`SimplyTheBest` applies the definitions in Sections 3.2 and 3.3 to the concrete functions induced by instruction patterns in LLVM IR. Dealing with such patterns moves the problem from synthesizing a transformer for an isolated operation to synthesizing a transformer for a loop-free sequence.

We assume a fixed instruction set $\mathcal{I} = \{op_1, \ldots, op_n\}$, where each primitive instruction op_i has a concrete semantics and an existing sound abstract transformer $op_i^\sharp$. A pattern is built from these primitive instructions, but denotes a single concrete function from the free variables of the pattern to its output.

Definition 3.8 (Pattern). A *pattern* is an acyclic static single-assignment (SSA) fragment whose internal nodes are instructions, whose leaves are variables, and whose distinguished root is the value produced by the fragment. The free variables of the fragment determine the pattern inputs, and the root determines its output. Thus, a pattern with k free variables denotes a concrete function $pat : C^k \to C$, and an abstract transformer for that pattern is a function $pat^\sharp : \mathcal{A}^k \to \mathcal{A}$.

For example, the expression `x -nsw (x % y)` from Section 2 corresponds to the pattern

$$\begin{aligned} r &= \mathrm{srem}(x, y) \\ o &= \text{sub-nsw}(x, r). \end{aligned} \tag{1}$$

This pattern has two free variables, `x` and `y`; one internal value, `r`; and a distinguished root, `o`. The corresponding concrete function is $pat(x, y) = x - nsw(x\%y)$, and a pattern transformer maps abstract facts about `x` and `y` to an abstract fact about `o`.

Definition 3.9 (Sequential Transformer). The *sequential transformer* for a pattern pat, written $pat^\sharp_{\text{seq}}$, is the abstract transformer obtained by evaluating the pattern in dependency order: each free variable is assigned the corresponding abstract input, each instruction node labeled by op is evaluated using the best per-instruction transformer $op^\sharp$ on the abstract values of its operands, and the transformer returns the abstract value computed for the distinguished root.

For the pattern in Equation (1), the sequential transformer has the same SSA shape, but replaces concrete values with abstract values and primitive instructions with their abstract transformers:

$$\begin{aligned} R &= \mathrm{srem}^\sharp(X, Y) \\ O &= \text{sub-nsw}^\sharp(X, R). \end{aligned} \tag{2}$$

Thus $pat^\sharp_{\text{seq}}(X, Y) = O$. In the terminology of Section 3.3, $pat^\sharp_{\text{seq}}$ is the base transformer for the pattern. The goal of specialization is to augment this base transformer with a lookup table on the profiled inputs.

Combining multiple patterns. `SimplyTheBest` builds an analyzer with transformers for a collection of patterns $\mathcal{P}$. When queried with an SSA fragment (which is itself a pattern), `SimplyTheBest` applies the transformers of *all* matching patterns in $\mathcal{P}$ and takes the meet of the results.

Definition 3.10 (Pattern matching). Let $pat_1 : C^m \to C$ and $pat_2 : C^n \to C$ be patterns with m and n free variables, respectively. We say pat_1 *matches* pat_2, written $pat_1 \preceq pat_2$, if there exists a substitution $\theta_{pat_1 \preceq pat_2}$ mapping each of the m free variables of pat_1 to a term over the free variables of pat_2 such that $pat_1 \circ \theta_{pat_1 \preceq pat_2} = pat_2$. In this case we also call pat_1 a *subpattern* of pat_2.

For example, let pat be the pattern in Equation (1) with free variables x and y, and let D be the following queried SSA fragment/pattern with free variables x', y', and z',

$$\begin{aligned} w &= \text{add}(y', z') \\ r &= \text{srem}(x', w) \\ o &= \text{sub-nsw}(x', r). \end{aligned} \tag{3}$$

Then pat matches D (i.e., $pat \preceq D$) under the substitution $\theta_{pat \preceq D} = \{\, x \mapsto x', \quad y \mapsto \text{add}(y', z') \,\}$.

After matching a pattern, `SimplyTheBest` recursively analyzes the sub-fragments to which the pattern's free variables are mapped—namely, the substitution terms x' and $\text{add}(y', z')$ in this example. At each recursive step, `SimplyTheBest` matches those sub-fragments against other patterns, applies the corresponding transformers, and continues until it reaches the leaves of D. In the following definition, we write $pat(x_1, \ldots, x_k)$ to denote a pattern with free variables $x_1, \ldots, x_k$.

Definition 3.11 (Semantics of `SimplyTheBest`*).* Let D be a query pattern and $\mathcal{P}$ a collection of patterns, where each pattern $pat \in \mathcal{P}$ has an associated transformer $pat^\#$. The semantics of `SimplyTheBest`'s analysis ($\mathtt{STB}_\mathcal{P}$) is defined as follows:

$$[\![\mathtt{STB}_\mathcal{P}]\!](D) := \bigsqcap_{\substack{pat(x_1,\ldots,x_k) \in \mathcal{P} \\ pat \preceq D}} pat^\#\big([\![\mathtt{STB}_\mathcal{P}]\!](\theta_{pat \preceq D}(x_1)),\ \ldots,\ [\![\mathtt{STB}_\mathcal{P}]\!](\theta_{pat \preceq D}(x_k))\big).$$

The base case assigns top to free variables: $[\![\mathtt{STB}_\mathcal{P}]\!](x) = \top$.

In practice, every $pat^\#$ is a lookup-table transformer that is specialized to the set of specific abstract values needed by the analysis. We also assume every single instruction forms a one-instruction pattern in $\mathcal{P}$, keeping its existing transformer, so the recursion can always descend through each instruction down to the leaves of the SSA fragment.

Cross-pattern pruning. Once multiple patterns are involved, an entry of the table may become redundant because it is subsumed by the transformer of one of its subpatterns (Definition 3.10).

For example, the pattern in Equation (1), $pat = \text{sub-nsw}(x, \text{srem}(x, y))$, has as a single-instruction subpattern $pat_a = \text{sub-nsw}(x', y')$; the relation $pat_a \preceq pat$ holds under the substitution $\theta_{pat_a \preceq pat} = \{\, x' \mapsto x, \quad y' \mapsto \text{srem}(x, y) \,\}$.

If an entry in pat's table does not rely on the information given by the `srem` operator and the reuse of x, then that entry may already be subsumed by pat_a's transformer. For example, suppose pat's table contains another entry $\mathbf{E4} = \langle (x{=}\texttt{??00},\ y{=}\texttt{1100}) \mapsto \texttt{??00} \rangle$ recording that $\text{sub-nsw}(x, \text{srem}(x, y))$ has its low two bits zero when $x = \texttt{??00}$ and $y = \texttt{1100}$. This precision does not actually come from the `srem` node or from the reuse of x: it holds simply because both operands of the outer subtraction have their low two bits zero.

If we push $\mathbf{E4}$'s input through the substitution $\theta_{pat_a \preceq pat}$ to obtain the induced input for pat_a, the first operand is $x' = x = \texttt{??00}$ and the second is $y' = \text{srem}^\#_{\text{seq}}(\texttt{??00}, \texttt{1100}) = \texttt{??00}$—i.e., the remainder of a multiple of four by a multiple of four is again a multiple of four. If pat_a's table

happens to contain an entry with the same input ($x' = \texttt{??00}, y' = \texttt{??00}$) and the same output `??00`, entry **E4** is redundant and `SimplyTheBest` removes it from *pat*'s table.

Such a cross-pattern subsumption can be formally defined by lifting the substitution terms to their sequential transformer $\theta^{\#}$:

Definition 3.12 (Cross-subsumed Entry). Let *pat* and pat_a be patterns such that $pat_a \preceq pat$, and let $pat^{\#} = (f^{\#}_{pat}, T_{pat})$ be the lookup table transformer for *pat* and $pat^{\#}_a$ be the transformer for pat_a. Assume $f^{\#}_{pat}$ and $pat^{\#}_a$ are monotone.

We say that $\langle \vec{A} \mapsto O \rangle \in T_{pat}$ is *cross-subsumed* by $pat^{\#}_a$ if $pat^{\#}_a\big(\theta^{\#}_{pat_a \preceq pat}(\vec{A})\big) \sqsubseteq O$.

Cross-subsumed entries can be safely pruned from tables. Definition 3.11 makes it clear why doing so does not affect the analysis: whenever a removed entry from a specific pattern would have applied, the more general pattern also matches, and its transformer is at least as precise.

No reliance on LLVM's transformers. The table construction in this section is self-contained: it assumes the baseline per-instruction transformers are optimal (Definition 3.9) and focuses solely on the precision gained by reasoning about patterns as a whole. Whenever the construction requires evaluating the per-instruction transformer on an input, `SimplyTheBest` computes the best output instruction-by-instruction with an SMT solver (Section 3.3.1), rather than calling LLVM's transformers. As a result, `SimplyTheBest` never has to locate those transformers, which are scattered throughout LLVM's source code. The optimality assumption is also empirically reasonable for LLVM: Peng et al. [42] found that most of LLVM's transformers for `KnownBits` and `ConstantRange` are maximally precise. If one were to apply `SimplyTheBest` to domains that lack optimal per-instruction transformers, the same table-construction technique could also be applied to the transformers themselves. A further consequence is that the monotonicity assumptions in Theorem 3.7 and Definition 3.12 hold automatically, because best transformers and lookup-table transformers are always monotone.

4 From LLVM Profiles to Best Local Pattern Transformers

This section instantiates Section 3 for LLVM's demand-driven integer analyses. Figure 2 sketches the common code skeleton of LLVM's analyses, and shows where we add instrumentation—at the end of the existing analysis functions.

For each domain (`KnownBits`, `UConstRange`, `SConstRange`), `SimplyTheBest` first profiles calls to `ComputeKnownBits()` and `ComputeConstantRange()` over a corpus of LLVM IR (Section 5) to mine a set of candidate SSA pattern DAGs, which we write $\mathcal{P}_{\text{raw}}$. It then filters and reduces these candidates using the completeness criteria from Section 4.2, producing a smaller set of canonical patterns $\mathcal{P}$. Finally, for each $pat \in \mathcal{P}$, `SimplyTheBest` collects the profiled abstract inputs $\mathcal{S}_{pat}$ and builds a compressed lookup-table transformer T_{pat}, to enhance the instruction-by-instruction analysis in LLVM. In `SimplyTheBest`'s pipeline, LLVM's analyses will be instrumented and run three times: once to mine patterns, once to profile each pattern's abstract inputs, and once to deploy and measure the impact of the synthesized transformers.

4.1 Mining Frequent Patterns

Given a corpus of LLVM IR and a target analysis, the first step of `SimplyTheBest` is to discover the set of instruction patterns $\mathcal{P}_{\text{raw}}$ that the analysis actually encounters. The input to this phase is the corpus, the LLVM analysis being improved—`ComputeKnownBits()` for `KnownBits` or `ComputeConstantRange()` for ranges—and the set of LLVM operations that `SimplyTheBest` supports. The output is a set of instruction directed acyclic graphs (DAGs), each paired with the number of times the DAG appears in analysis queries. These counts estimate how often a

```
Domain computeDomain(Node *v, unsigned Depth) {            // Domain: e.g. KnownBits, ConstantRange
  if (Depth > depth_limit || isBlocker(v)) return Domain::Top;
  if (isConstant(v)) return Domain::alpha(getConstant(v));

  Domain result = Domain::Top;
  if (match(v, Add))                                  // LLVM built-in per-instruction transformer
    l, r = getOperands(v, Add);
    out = Add#(computeDomain(l, Depth + 1), computeDomain(r, Depth + 1));
    result = Domain::meet(result, out);
  // ... other primitive instructions ...

  // Pattern transformers deployed here
  return result;
}
```

Fig. 2. SimplyTheBest's model of LLVM's demand-driven analyses over an abstract domain Domain. It computes the abstract value of an SSA value by recurring on its operands—applying a built-in transformer such as Add# at each instruction and meeting the results in Domain—until it reaches a constant, a blocker, or the depth limit. SimplyTheBest deploys its synthesized pattern transformers just before returning; the deployment hook is illustrated in Figure 3.

synthesized transformer for that DAG might be applied during the analysis; the final number of deployed queries can differ, as Section 2 shows, because added precision may enable LLVM to fire optimizations that rewrite the IR, changing the number of matches.

A primer on LLVM analyses. LLVM's lightweight integer analyses are demand-driven. Rather than eagerly running a global fixpoint computation over the whole function, they answer queries about individual SSA values. The analysis recurs backward through the SSA graph (ConstantRange does not recur), querying operands of supported instructions until it reaches a constant, an unsupported operation, a function boundary, or LLVM's recursion limit. Constants are abstracted directly into the corresponding element in the abstract domain; the other stopping cases return $\top$. The analysis then propagates information back toward the queried root—i.e., the SSA value whose abstract value was originally requested—by applying LLVM's existing per-instruction abstract transformers. Thus, each query exposes a finite backward slice of IR that LLVM is willing to inspect for that root value.

Slicing analysis queries. To mine patterns, SimplyTheBest instruments the target analysis function with a hook. The mining hook records the slice exposed by the query but does not change the value returned by the analysis.

For every query, including recursive queries triggered by a parent query, SimplyTheBest builds the instruction DAG rooted at the queried SSA value and bounded by the query's remaining recursion budget. The slicer expands only operations in the subset that SimplyTheBest supports, and stops at the same boundaries as the underlying LLVM analysis, so the mined DAG has the same context as the original query. It also preserves SSA sharing by assigning one node to each LLVM value; this is essential for exposing patterns with SSA reuse such as Equation (1). Each sliced DAG is logged, with its corresponding frequency count.

Separating target domains. The same slicing procedure is used for both KnownBits and ConstantRange, but range queries also carry the ForSigned flag, indicating whether the client wants a result interpreted using signed or unsigned order. Since SimplyTheBest models LLVM

ranges using the two lattice domains `SConstRange` and `UConstRange`, it keeps these profiles separate: queries with `ForSigned` set contribute to the `SConstRange` pattern set, and the remaining queries contribute to the `UConstRange` pattern set. Thus each mined DAG is associated not only with a frequency count, but also with the abstract domain in which its transformer will be synthesized.

Pattern schemas and supported operations. `SimplyTheBest` supports all of LLVM's arithmetic and bitvector operations, as well as many LLVM intrinsics. Since LLVM's arithmetic and bitvector operations such as `add` may be instantiated at any bitwidth, `SimplyTheBest` treats these cases as instances of the same pattern schema. Thus a mined pattern such as Equation (1) can be instantiated at any bitwidth. To include the useful Boolean-producing/consuming operations `ICmp` and `select`, `SimplyTheBest` allows pattern inputs, outputs, and internal nodes to have type either `i1` or `iN`, for some fixed bitwidth N. `SimplyTheBest` also includes the casts `Zext:i1->iN`, `Sext:i1->iN`, and `Trunc:iN->i1`. This type scheme is general enough to capture mixed Boolean and bitvector idioms, but restrictive enough to avoid exploding the number of distinct mined patterns.

Operations with overflow or other semantic flags are treated as distinct operators rather than as annotations on a single operator. For example, `Add`, `AddNsw`, `AddNuw`, and `AddNswNuw` are four separate operators. This choice makes the mined DAGs faithful to the semantics that LLVM's analyses actually use: a flagged operation may be able to determine dataflow facts that the unflagged operation cannot. The full list of supported operators and their semantics appears in Appendix B.

The result of this phase is a domain-indexed set of instruction patterns, each with a frequency count. These patterns are still raw slices of LLVM analysis queries: they may contain complete edges that cannot improve precision, and they may appear in syntactically different but match-equivalent forms. The next phase reduces the pattern set.

In practice. Across all IR code in `llvm-opt-benchmark` (558 million instructions), we observed 1.53 B calls for `KnownBits`, 430 M calls for `UConstRange`, and 165 M calls for `SConstRange`. These calls involved millions of abstract values. Out of all the calls, we mined thousands of unique patterns: 421 K `KnownBits`, 131 K `UConstRange`, and 71.5 K `SConstRange`.

4.2 Pattern Reduction: Filtering Out Sequentially Complete Subpatterns

Intermediate computations in a pattern may not add to the precision gap between the sequential and composite transformers. We provide a cheap and sound check for finding such computations. The basic idea is to identify cases in which a sequential transformer does not lose precision at an intermediate abstract value, given the abstract domain at hand.

In the sequential transformer for our running pattern (Equation (2)), the intermediate abstract value is R. A transformer for the whole pattern can reason directly about the relationship between $\mathrm{srem}(x, y)$ and the final subtraction. The sequential transformer, in contrast, must summarize the possible values of the remainder as R before analyzing sub-nsw, possibly losing precision. Formally, the compositional precision loss that `SimplyTheBest` targets occurs when the exact set of possible remainders $C_R = \widetilde{\mathrm{srem}}(\gamma(X), \gamma(Y))$ is replaced by its abstraction and concretization, $\gamma(\alpha(C_R))$. The sequential transformer then analyzes sub-nsw using this closed set rather than the exact set C_R. For instance, when y = `??00`, every concrete value of `y` is a multiple of 4. The whole-pattern transformer can use this fact to conclude that `x -nsw (x % y)` has zeros in its lowest two bits. Sequential reasoning may lose this fact after abstracting the intermediate value `x % y`.

We now state the sufficient condition used by `SimplyTheBest` to decide when such an intermediate abstraction is harmless. Consider an arbitrary producer–consumer pair:

$$\begin{aligned} r_1 &= f(\vec{x}) \\ r_2 &= g(r_1, \vec{y}). \end{aligned} \tag{4}$$

In Equation (4), sequential reasoning must abstract the possible values of r_1 before analyzing g. We write $\widetilde{f}$ and $\widetilde{g}$ for the collecting semantics of the corresponding concrete operations—i.e., $\widetilde{f}(\vec{C}) = \{ f(\vec{c}) \mid c_i \in C_i \}$.

The notions of forward and backward completeness by Giacobazzi and Quintarelli [11] identify cases where composing sequential transformers does not result in loss of precision. Forward completeness says that the producer already produces an intermediate set that is exactly representable by the abstract domain, so $\gamma(\alpha(C)) = C$. Backward completeness says that, even if $\gamma(\alpha(C)) \supset C$, the consumer cannot observe the difference after abstraction of the final result.

Definition 4.1 (Forward and Backward Completeness). An operation op is *forward complete* for the abstract domain $\mathcal{A}$ if, for every abstract input $\vec{A} \in \mathcal{A}^k$, we have $\gamma(\alpha(\widetilde{op}(\gamma(\vec{A})))) = \widetilde{op}(\gamma(\vec{A}))$.

An operation op with n arguments is *backward complete in argument i* for $\mathcal{A}$ if, for every concrete set $C \subseteq \mathcal{C}$ in position i and other abstract values $A_j \in \mathcal{A}$ for $j \neq i$:

$$\alpha(\widetilde{op}(\gamma(A_1), \ldots, \gamma(\alpha(C)), \ldots, \gamma(A_n))) = \alpha(\widetilde{op}(\gamma(A_1), \ldots, C, \ldots, \gamma(A_n)))$$

For a unary operation, this reduces to $\alpha(\widetilde{op}(\gamma(\alpha(C)))) = \alpha(\widetilde{op}(C))$.

Corollary 4.2 (Completeness of Best Sequential Composition). *Consider the producer–consumer pattern in Equation (4), in which no SSA value is used more than once. Assume the primitive transformers for f and g are best abstract transformers. Let $pat^{\#}_{\text{best}}$ be the best abstract transformer for the whole pattern, and let $pat^{\#}_{\text{seq}}$ be the sequential composition of the primitive best transformers. If f is forward complete for $\mathcal{A}$, or if g is backward complete for its r_1 argument, then $pat^{\#}_{\text{best}} = pat^{\#}_{\text{seq}}$.*

This follows from Lemma 3.4 of Giacobazzi and Ranzato [12], who also show that the exact condition is local backward completeness of g on the image of f, that is, on the concrete sets the producer can actually emit. That condition depends on the producer and consumer pair together, so it must be established separately for each pattern, whereas forward and backward completeness of a single operation can be proved once and reused across all patterns in which the operation appears.

Mechanization. We have formalized Definition 4.1, Corollary 4.2, and the completeness criteria for `KnownBits`, `UConstRange`, and `SConstRange` in Lean. Using these characterizations we classify 63 LLVM IR operations as forward-complete, backward-complete, or both in each domain. Completeness of each operation is proved for all bitwidths, and each incomplete operation has a concrete counterexample at a fixed width. `SimplyTheBest` consults this classification during pattern reduction and collapses any producer–consumer pair in which f is forward complete or g is backward complete for the argument receiving r_1, since by Corollary 4.2 specialization cannot improve on sequential composition there. Appendix A gives the domain-specific statements of the criteria and the full classification tables.

4.2.1 Pattern Reduction. We lift the operation-level notions of forward and backward completeness from Definition 4.1 to patterns. A use of an SSA value v by an instruction op is *complete* if either the producer of v is forward complete or the consumer op is backward complete at v. Otherwise, the use is *incomplete*.

Pattern reduction is the process that *cuts* the mined patterns into smaller, incomplete subpatterns. Because matching against large patterns is a source of overhead in practice, pattern reduction is an important step in reducing matcher cost while maintaining precision. A pattern from the initial set $\mathcal{P}_{\text{raw}}$ may be built fully or partially from complete SSA uses, in which case the pattern is larger than necessary to realize all available precision gains. Reduction extracts subpatterns from the raw patterns by cutting complete SSA uses and retaining the regions that may still improve precision. Because completeness is a per-use criterion, it does not account for relationships among distinct

uses. In particular, when an SSA value has multiple uses, every use carries the same concrete value. A cut separating those uses loses that fact, which the composite transformer can exploit, so `SimplyTheBest` never cuts between uses of a single SSA value.

Consider an internal SSA use in which an instruction g consumes the value r produced by an instruction f, as in Equation (4). If it is complete, the use is *cuttable* when r has no other uses and the subpattern rooted at f shares no pattern input with the rest of the pattern. Cutting splits the pattern into a *producer side* ρ_{prod}, the subpattern rooted at f, and a *consumer side* ρ_{con}, the rest of the pattern with r replaced by a fresh input. These observations motivate the following definition of a pattern reduction phase:

Definition 4.3 (Pattern reduction). Pattern reduction simplifies a set of patterns $\mathcal{P}_{\text{raw}}$ to $\mathcal{P}_{\text{ref}}$, recursively cutting each pattern and filtering trivial single-instruction patterns:

$$\text{reduced}(pat) = \begin{cases} \emptyset, & pat \text{ is one instruction with no repeated operand,} \\ \{pat\}, & pat \text{ is not cuttable,} \\ \text{reduced}(\rho_{prod}) \cup \\ \quad \text{reduced}(\rho_{con}), & \text{otherwise} \end{cases}$$

$$\mathcal{P}_{\text{ref}} = \bigcup_{pat \in \mathcal{P}_{\text{raw}}} \text{reduced}(pat).$$

Reduction differs by domain because LLVM's two analyses differ in how they issue queries. `ComputeKnownBits()` recursively analyzes the operands of the value it is asked about, up to a depth limit, so every interior instruction of a pattern is itself queried. `ComputeConstantRange()` does not recur, so only the value the client asks about is ever matched. A subpattern without the queried root would not be able to fire, so reduction in the constant range domains removes only producers that reduce to nothing, ensuring that every member of reduced(pat) contains the original root.

Note that reduction never grows the pattern set. In `KnownBits`, a member without the root is subsumed by a pattern that was already mined at that interior value, since `ComputeKnownBits()` queries every interior value. In `UConstRange` and `SConstRange` every member of reduced(pat) contains the root and so replaces pat. Reduction is also idempotent.

We illustrate reduction with three examples in the `KnownBits` domain. Green variables mark complete SSA references, and red variables are either incomplete or involved in SSA value reuse.

Cutting a complete producer: In the pattern below, the use of b in the computation of o is complete, as is the use of a in the computation of b. The reduced pattern therefore cuts away the computation of b and replaces its value with a fresh argument z:

Original pattern	$\rightarrow$	Reduced pattern
$a = u \,\&\, v$		
$b = w \oplus a$		
$c = x - y$		$c' = x' - y'$
$o = b + c$		$o' = z + c'$

Discarding the root: The following pattern contains incomplete uses, but the root operation $\oplus$ cannot exploit the precision losses inside either branch because it is backward complete in `KnownBits`. So `SimplyTheBest` replaces the pattern with its two incomplete subpatterns, each of

which was already mined when the analysis recurred into b and d:

Original pattern	⇉	Reduced pattern 1	Reduced pattern 2
$a = x \div y$		$a' = x' \div y'$	
$b = a - z$		$o' = a' - z'$	
$c = u \times v$			$c' = u' \times v'$
$d = c + w$			$o' = c' + w'$
$o = b \oplus d$			

Reuse blocks a cut: Here & is forward complete, so the use of a would be cuttable, except that the input x feeds both a and o. Cutting would separate two uses of x, so the pattern is kept as-is:

$$a = x \,\&\, y$$
$$o = a \oplus x$$

4.2.2 Canonicalization. Before collecting abstract inputs, `SimplyTheBest` canonicalizes patterns in $\mathcal{P}_{\text{ref}}$ according to the equivalences recognized by LLVM's embedded pattern-matching DSL. LLVM uses this DSL to recognize IR fragments and select the appropriate abstract transformers; if two patterns differ only in a way that LLVM's matcher would treat identically, `SimplyTheBest` keeps a single canonical representative. `SimplyTheBest` does not perform associative rewrites or algebraic simplifications during canonicalization.

There are three forms of canonicalization:

- *Commutative operators.* Children of commutative operators are canonicalized recursively, then sorted by canonical subtree text.
- *ICmp predicates.* Comparisons are normalized to use a common predicate direction; for example, `ICmpUgt(a, b)` is rewritten as `ICmpUlt(b, a)`.
- *Alpha renaming.* Argument names are canonicalized by first use in the final canonical traversal, since LLVM matchers do not depend on source-level argument names.

In practice. A small fraction of patterns are composed entirely of complete SSA uses: 4.64 % of `KnownBits`, 3.69 % of `UConstRange`, and 8.13 % of `SConstRange`. Producer-cut reduction affects 21.3 % of `KnownBits` patterns, 15.1 % of `UConstRange`, and 19.7 % of `SConstRange` patterns. Root-split reduction affects 18.6 % of `KnownBits`, 15.3 % of `UConstRange`, and 22.8 % of `SConstRange`. Together with canonicalization, they remove 120 K patterns in `KnownBits`, 27 K in `UConstRange`, and 21 K in `SConstRange`.

4.3 Input Specialization and Table Pruning

The previous phase produces a set of reduced, canonicalized patterns. The final phase turns each such pattern into a best local transformer, realized as a table that maps abstract inputs to precise outputs (Definition 3.2).

To do so, `SimplyTheBest` collects the abstract inputs on which the pattern is queried, computes maximally precise outputs for those inputs, removes entries already subsumed by the base transformer or by other tables, and emits LLVM matchers that implement the lookup-table semantics from Section 3.3.2. Removing redundant entries is essential for deployment: each retained entry adds matching and coverage checks to a pervasive compiler analysis, so table compression directly limits compile-time overhead.

Profiling abstract inputs. The first profiling pass discovers pattern shapes; a second profiling pass collects the profiled input set $\mathcal{S}_{pat}$ for each pattern pat. To collect this set, `SimplyTheBest` instruments LLVM's analysis again, deploying for each pattern a *stub transformer* that observes inputs. Figure 3 shows the deployment hook we patch to LLVM's analysis in Figure 2: whenever

```
// deployed inside computeDomain, after the built-in transformers
if (match(v, pat1))                                          // match a mined pattern
  arg0, ..., argn = getPatternOperands(v, pat1);
  d0, ..., dn = getOperandEffectiveDepth(pat1);              // shortest leaf-to-root distance
  out = pat1#(computeDomain(arg0, Depth + d0), ...);
  result = Domain::meet(result, out);
// ... other patterns ...
```

Fig. 3. SimplyTheBest's deployment hook, added to computeDomain (Figure 2). On a match, it (1) gets the pattern's leaf operands, (2) computes each leaf's effective depth—its distance to the pattern root, so recursion still charges the depth budget for the collapsed instructions—and (3) applies the pattern's transformer pat1# to the operands' abstract values and meets the result into Domain. The actual implementation matches all mined patterns together in a shared tree, rather than scanning linearly as shown above.

ComputeKnownBits() or ComputeConstantRange() matches the pattern, the stub records the abstract input vector supplied to the pattern. This produces the set of abstract inputs on which the best local transformer for *pat* must be optimal. The stub is used only during profiling; during deployment it is replaced by the synthesized lookup-table transformer.

Constructing and compressing tables. For each profiled input $\vec{A} \in \mathcal{S}_{pat}$, SimplyTheBest computes the maximally precise output $pat^{\#}_{\text{best}}(\vec{A})$ using the SMT procedure from Section 3.3.1. It also computes the sequential output $pat^{\#}_{\text{seq}}(\vec{A})$, where $pat^{\#}_{\text{seq}}$ is the base transformer for the pattern. With these outputs on hand, SimplyTheBest builds a raw lookup table for each pattern,

$$T_{pat} = \{\, \langle \vec{A} \mapsto pat^{\#}_{\text{best}}(\vec{A}) \rangle \mid \vec{A} \in \mathcal{S}_{pat} \text{ and } pat^{\#}_{\text{best}}(\vec{A}) \neq pat^{\#}_{\text{seq}}(\vec{A}) \,\}. \tag{5}$$

The inequality filters out entries that are trivially subsumed (Def. 3.6) by the base transformer.

SimplyTheBest then applies to $(pat^{\#}_{\text{seq}}, T_{pat})$ the generic subsumption rule of Definition 3.6 to remove entries that are redundant within the table T_{pat}, and the cross-pattern subsumption rule of Definition 3.12 to remove entries that are redundant across tables.

All SMT reasoning is performed offline. During compilation, the generated code performs only pattern matching, coverage tests, table lookup, and lattice meets.

In practice. At this step, SimplyTheBest has several candidate transformers (tables) each with a set of abstract inputs and the corresponding SMT-computed precise abstract outputs. The output is often top (61.07 % of KnownBits inputs, 56.59 % UConstRange, and 63.51 % SConstRange) and we discard the corresponding entries from the table. Then we discard entries whose sequential output is already as precise as the composite one (Equation (5)), which arises about one-third of the time (32.66 % KnownBits, 35.79 % UConstRange, 28.82 % SConstRange). We also drop inputs where SMT times out (0.04 % KnownBits, 0.26 % UConstRange, 0.67 % SConstRange) and where the output is bottom. We might gain precision by signaling a conflict when the output is bottom, but this affects only 21 rows and few consumers in LLVM are equipped to handle a conflict value. Finally, we prune entries both intra-table (Definition 3.6) and cross-table (Definition 3.12). This step further removes about 2 % of entries in each domain.

On average, the table component of a lookup-table transformer consists of 12 rows in KnownBits, 8 rows in UConstRange, and 4 rows in SConstRange. There are outliers, however. In KnownBits, the biggest table has 2,172 rows. The other domains have smaller maxima: 614 rows in UConstRange and 502 rows in SConstRange.

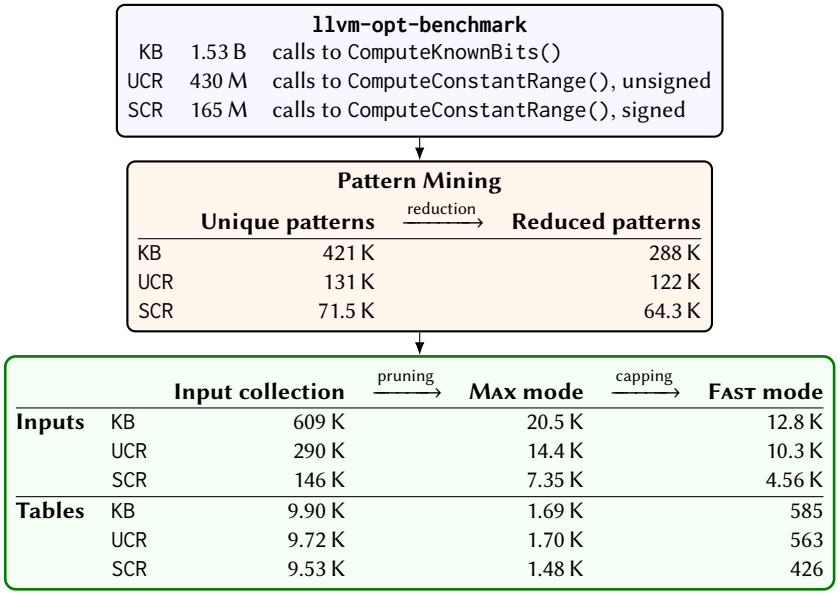


Fig. 4. Narrowing millions of observed inputs to thousands stored in `SimplyTheBest`.

5 Experiment Setting and Implementation Details

Our implementation (and quantitative evaluation) is based on `llvm-opt-benchmark`, a dataset of LLVM IR designed to guide compiler optimization work [54]. It is maintained by the LLVM developer community and runs nightly to assess the impact of changes to LLVM. The code in the benchmark is the compiled output from 240 different programs, originally written in C, C++, and Rust, that span domains such as simulation (Bullet), media codecs (FFmpeg), compression algorithms (zlib, zstd), libraries (Boost, Abseil), systems code (the Linux kernel), interpreters (CPython, Ruby), and compilers (LLVM itself, V8). It consists of 558 million LLVM instructions and 38,729 files.

We use `llvm-opt-benchmark` both for mining patterns and to test precision gains after constructing `SimplyTheBest`. When mining patterns, we start with the unoptimized versions of all programs in the benchmark. We optimize this code using our fork of LLVM 22.1.4 that intercepts calls to `ComputeKnownBits()` and `ComputeConstantRange()` for profiling and transformation. The rest of the LLVM optimization pipeline remains unchanged.

After construction, `SimplyTheBest` stores just a few thousand abstract inputs to guide compilation. To gain an understanding of the tradeoffs between specialization and compilation overhead, we evaluated two out of a very large number of possible configurations:

- Max mode of `SimplyTheBest` aims for the highest precision. Across all transformers, it stores 20,473 `KnownBits` inputs, 14,428 `UConstRange` inputs, and 7,347 `SConstRange` inputs.
- Fast mode reduces compile-time overhead by omitting any transformer that, during training, improved fewer than 100 queries. It also limits the number of transformers that share a common root (i.e., outermost SSA operation) to 25 total, to speed up pattern disambiguation. Across all transformers, it stores 12,778 `KnownBits` inputs, 10,299 `UConstRange` inputs, and 4,559 `SConstRange` inputs.

Figure 4 illustrates just how sparse "a few thousand inputs" is by outlining the path from source-code mining to transformer construction. Starting from the 558 million instructions in

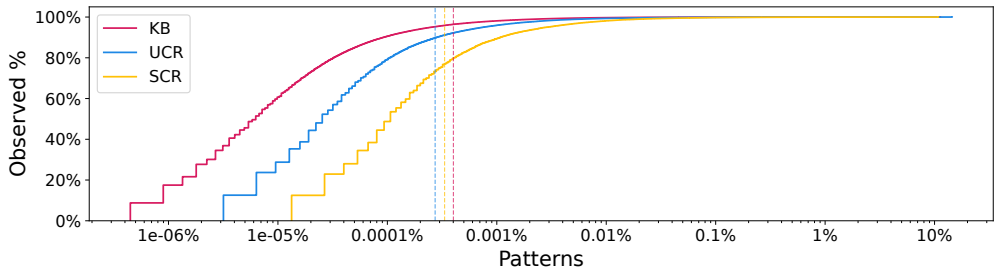


Fig. 5. Cumulative distribution function of pattern frequency. The x axis counts the percentage of patterns (log scale), sorted by their rank on the y axis. The y axis shows how often a pattern was observed during training. The vertical lines show our cutoff at 10 K patterns for each domain.

`llvm-opt-benchmark` and several billion calls to compute static information, mining of calls uncovers merely thousands of patterns. Pattern reduction removes up to half of these. We select the top 10 K patterns for further analysis, eliminating a long tail of infrequently observed patterns. Finally, table pruning drastically reduces the number of abstract inputs, eliminating over 93 % of rows in the transformer tables. Max mode uses exactly these abstract inputs.

Pattern Selection. Many of the patterns that pass reduction are relatively uninteresting because they appear infrequently during training. This can be seen through Figure 5, which shows how often each pattern occurred. The top 10 K patterns account for at least 70 % of all observations; we use this set of patterns for input specialization and table pruning, and discard the rest. With the selected patterns, the final steps of table construction take approximately 12 hours using a 64-core machine.

Fewer than 10 K patterns from each domain proceed to the table construction step. The exact numbers are 9.90 K patterns for `KnownBits`, 9.72 K for `UConstRange`, and 9.53 K for `SConstRange`. Some patterns only see non-integer inputs, such as `Select(x, Select(y, a, b), c)` where a, b, and c are all pointers, which we do not support. In the `ConstantRange` domains, we enforce a limit of 500 patterns per root because many patterns begin with the same operation, which causes slowdowns in matching code. Finally, a small number of patterns (roughly 10) are long chains of `or` and `lshr` that we do not support due to their extreme length.

6 Evaluation

We look at optimizations `SimplyTheBest` enables (Section 6.1), and then empirically assess its soundness using random program generators (Section 6.2). Next, we study its precision gains and compilation overhead (Section 6.3). When "re-trained" on a subset of `llvm-opt-benchmark` and tested on the remaining code, `SimplyTheBest` still performs well, recovering approximately 90 % of the `KnownBits` precision gain (Section 6.4).

Experiment Setting. Experiments ran on a single-user desktop running Ubuntu Server 26.04 with 10 CPU cores (Intel i7-6950X at 3.00 GHz) and 64 GB memory, unless otherwise noted. When measuring compile times, we follow the LLVM Benchmarking Guide [32]. This includes making a release build of LLVM, pinning all processes to individual cores, and disabling hyperthreading, turbo boost and kernel ASLR. To collect compile times, we ran the compiler three times and took the fastest one.

6.1 Optimizations Unlocked Through SimplyTheBest

We looked at each use of known bits in LLVM's middle-end optimization passes, and instrumented a subset of them where we could see a clear causal relation between a known bits fact and a subsequent program transformation. (Other uses of known bits, for example in LLVM's alias analysis, did not lead to any sort of short causal chain that we could trace.) We then optimized `llvm-opt-benchmark` using both the baseline version of LLVM and `SimplyTheBest` (Max mode, Section 5), in order to assess the differences. Out of the 165 optimizations that we instrumented, 59 of them fired more often due to our changes and 12 fired fewer times. Across the benchmark, optimizations that we instrumented fired a total of 34,207 more times due to `SimplyTheBest`. One notable example, a transformation to drop a redundant `llvm.assume` intrinsic, fired 15 times more often (2,068 instances vs. 139) due to our changes.

We were curious: specifically, what other kinds of extra optimizations have we unlocked? Towards this end, we used the `llvm-reduce` tool to minimize LLVM IR files from `llvm-opt-benchmark` under the criterion that the LLVM IR emitted by an unmodified `opt -O3` and the LLVM IR emitted by `SimplyTheBest`'s `opt -O3` are not the same. This had the effect of finding small LLVM files that contain code that we analyze more precisely than LLVM does and, furthermore, the precision feeds an actual optimization that could not otherwise be performed.

In many examples that we looked at, extra analysis precision simply permitted LLVM to infer an undefined-behavior-related instruction flag. In the transformation below, reduced from ClamAV, both the original version of LLVM's optimizer and also ours convert a 64-bit comparison into a 32-bit one, allowing the zero-extend instruction to be eliminated, and then additionally our version also marks the final zero-extension as `nneg`, making it return a poison value if its argument is negative. The non-negative information can be useful because it permits the zero-extend operation to be soundly replaced by a sign-extension operation, which may be more efficient on a target such as RISC-V. The specific abstract transformer that provides the extra precision matches on `Select(ICmpUlt(x, y), z, y)`, where $x = 4$. For this pattern the sequential transformer cannot conclude that any bit returned by the `select` instruction is known, whereas the composite transformer concludes that all but the bottom two bits are zero. Only the top bit being known zero is necessary to soundly add the `nneg` qualifier.

```
define i64 @f(i32 %0) {
  %2 = zext i32 %0 to i64
  %3 = icmp uge i64 %2, 4
  %ss = select i1 %3, i32 0, i32 %0
  %4 = zext i32 %ss to i64
  ret i64 %4
}
```

⟶

```
define i64 @f(i32 %0) {

  %2 = icmp ugt i32 %0, 3
  %ss = select i1 %2, i32 0, i32 %0
  %3 = zext nneg i32 %ss to i64
  ret i64 %3
}
```

Other optimizations have a more direct impact. In the next one, reduced from a file in CMake, a value loaded from memory turns out to not be needed. The pattern here was `Udiv(Add(a, b), a)` where $b = -1$. The best sequential transformer returns all bits unknown, but the best composite transformer finds all bits to be known zeros.

```
define i64 @f(ptr %0) {
  %2 = load i64, ptr %0
  %3 = sub i64 %2, 1
  %4 = udiv i64 %3, %2
  ret i64 %4
}
```

⟶

```
define i64 @f(ptr %0) {
  ret i64 0
}
```

To make sure this one was not an artifact of test case reduction, we traced it back to the CMake source code and, indeed, this computation appears there, although the fact that the subtraction is by one is not apparent until the optimizer brings this constant closer to the operation.

6.2 Empirical End-to-End Soundness

Although `SimplyTheBest` is designed to always return sound dataflow results, implementation errors are always a possibility. We were also worried that increasing dataflow precision might expose latent LLVM bugs—this compiler is a vast collection of C++ that has not at any time, to our knowledge, been free of miscompilation defects.

We used six hours on a 16-core Linux machine to check that our `clang -O3` did not miscompile C++ programs generated by YARPGen v2 [31]. More than half a million programs were tested during this period, with no observed miscompiles. We then repeated the exercise using YARPGen v1 [30] and Csmith [52], giving each of them six hours on the same machine. The findings were the same.

6.3 Overhead of Lookup Table Transformers

Table 1 presents the precision gain and compile-time cost of `SimplyTheBest` on all benchmark code using both Max mode and Fast mode. We report precision as the number of bits added over LLVM's baseline (as both an absolute count and percent improvement). For `ConstantRange`, bits are counted as $\log_2(hi - lo)$. We also count the number of queries to `ComputeKnownBits()` and `ComputeConstantRange()` for which `SimplyTheBest` improves precision relative to LLVM. For costs, we report the mean and median slowdown in compile time for each file relative to LLVM.

Baseline LLVM finds 3 billion known bits using `KnownBits`, 1.88 billion using `UConstRange`, and 50 million using `SConstRange`. The largest relative improvement is close to 6 % precision in `SConstRange`. Fast mode for `SConstRange` does well compared to Max mode, retaining about 80 % of the precision gains.

Compile-time overhead is low in the `ConstantRange` domains, but rather high in `KnownBits` with Max mode (a mean of about 4.4 %). Switching to Fast mode reduces the average overhead to about 1.1 %. Curiously, in both `ConstantRange` domains Fast mode appears slightly slower than Max mode in absolute time, and `SConstRange` in Max mode appears slightly faster than the baseline. We attribute the differences to noise; they are < 1 % and comparable to the sample-to-sample variation of our measurements, which is about 1 % per file.

Taken together, the results show that `SimplyTheBest` trades broad applicability for high precision on the queries it recognizes. This works well since LLVM's built-in `KnownBits` and `ConstantRange` analyses already provide the broad applicability, but struggle to gain very high precision on queries which lose a lot of information due to a lossy abstract domain. This point is further emphasized by observing the size of abstract outputs. On average, queries improved by `SimplyTheBest` produce smaller outputs than LLVM: 32.62 % smaller for `KnownBits` queries, 6.8 % smaller for `UConstRange`, and 40.27 % smaller for `SConstRange`.

Understanding Compile Times. Compiler users have conflicting requirements: they want fast compile times and they also want high-quality code generation (and other things too!). A heavily optimizing ahead-of-time compiler like LLVM at its `-O3` setting represents a somewhat extreme position, where users might be willing to wait for better results. We ran a few experiments to determine how much compile time is spent in the middle-end optimizers. For `sqlite3`, a 270,000-line database written in C that takes 23 seconds to compile using `clang -O3`, 50 % of that time is spent in the LLVM middle end optimizers. For `exprtk`, a 47,000-line math library written in C++ that takes 115 seconds to compile using `clang++ -O3` on the same machine, 41 % of the time is spent in the LLVM middle end optimizers. The overhead numbers that we have presented apply to

Table 1. Precision gain and compile-time overhead. Bits Added is the precision gain in millions of fractional bits (and percent improvement relative to LLVM). Improved Queries counts the number of calls where `SimplyTheBest` added precision (and percent improvement vs. LLVM's transformers). Slowdowns are relative to LLVM's baseline compile times (the percent mean is a geometric mean).

		`KnownBits`		`UConstRange`		`SConstRange`	
Mode	Metric	Value	Gain	Value	Gain	Value	Gain
Max	Bits added	37.7 M	1.19 %	12.6 M	0.67 %	2.95 M	5.84 %
	Improved queries	3.64 M	0.24 %	8.63 M	2.01 %	1.02 M	0.66 %
	Mean slowdown	23.5 ms	4.39 %	2.82 ms	1.53 %	-2.71 ms	0.29 %
	Median slowdown	1.73 ms	1.56 %	0.75 ms	0.76 %	-0.33 ms	−0.25 %
Fast	Bits added	32.4 M	1.03 %	11.2 M	0.60 %	2.36 M	4.68 %
	Improved queries	3.34 M	0.22 %	8.46 M	1.97 %	1.01 M	0.61 %
	Mean slowdown	5.37 ms	1.11 %	5.11 ms	1.33 %	3.50 ms	0.68 %
	Median slowdown	0.30 ms	0.29 %	1.39 ms	1.06 %	0.68 ms	0.56 %

only the compiler middle-end. Thus, they will be diluted by roughly a factor of two in practical compilation use cases—or more, since real builds also spend time in the linker and in other steps.

6.4 Cross Validation on Subsets of the Corpus

One threat to the validity of our evaluation so far is that we have used `llvm-opt-benchmark` as both a *training set* and a *test set*. This raises the question of overfitting: `SimplyTheBest` may not generalize to other programs, even though the number of abstract inputs leveraged by `SimplyTheBest` is small. Since `llvm-opt-benchmark` is large (more than half a billion LLVM instructions from well over 200 open source projects) and was hand-picked by the LLVM community to reflect their optimization priorities, internal holdout validation is best, in our opinion, rather than seeking out codes that are not in the benchmark suite.

For the cross-validation, we rebuild `SimplyTheBest` (Max mode, `KnownBits` only) on subsets of the benchmark and report results on code that was not in the subset. Concretely, we use five groups that are roughly equal in size and test one group at a time, using two splitting settings:

- *By file*, across all files. A project's `.ll` files may be spread across training and test sets.
- *By project*. The test set contains only projects that were never seen during training.

For each test group, the remaining four groups serve as the training set: we mine patterns, profile abstract inputs, and synthesize transformers from them. We then deploy the resulting transformers on the held-out test group. The machine used was a single-user server with 64 Intel Xeon Platinum 8375C vCPUs (2.90 GHz). Table 2 presents our findings. For each splitting setting, the columns show the number of: files or projects in each group, bits added relative to LLVM, and improved queries relative to LLVM. Next to each precision gain, we compare it to the gain achieved by `SimplyTheBest` with the full training set. The *vs. full* numbers thus show how much information re-built `SimplyTheBest` was able to recover given less input data. We recover 93 % of `KnownBits` precision when splitting by file, and 68 % when splitting by project. Repeated cross-validations with different (randomly selected) splits vary slightly (within ±1 % of the recovery rates above). We conjecture the difference is due to LLVM's use of nondeterministic data structures at optimization level `-O3`, which we run three times when building `SimplyTheBest`. Another possible cause is SMT heuristics changing the number of timeouts in table construction.

Table 2. Five runs of cross validation of `KnownBits` results (Max mode), splitting `llvm-opt-benchmark` by file (left) and by project (right). Each row reports the number of files or projects that were hidden during training and the precision gain relative to LLVM on this held-out code. The *vs. full* numbers compare to the gain computed by `SimplyTheBest` with full training data.

	Split by file					Split by project				
Test	Held-out files	Bits added Value	Bits added vs. full	Improved queries Value	Improved queries vs. full	Held-out projects	Bits added Value	Bits added vs. full	Improved queries Value	Improved queries vs. full
1	8,064	5.51 M	92.13 %	569 K	93.19 %	47	3.54 M	69.19 %	329 K	64.57 %
2	7,744	8.62 M	95.14 %	867 K	95.36 %	48	6.61 M	87.99 %	522 K	85.82 %
3	7,711	7.26 M	94.89 %	637 K	93.91 %	48	3.50 M	71.11 %	357 K	82.75 %
4	7,716	7.08 M	93.97 %	622 K	92.65 %	48	5.65 M	79.03 %	567 K	74.85 %
5	7,494	6.07 M	87.99 %	553 K	89.27 %	49	5.87 M	47.20 %	884 K	74.89 %
Total	38,729	34.5 M	93.04 %	3.25 M	93.09 %	240	25.2 M	67.78 %	2.66 M	76.25 %

The gap between splitting by file and splitting by project is substantial. It suggests that the training data matters for making the most of `SimplyTheBest`: even though only a small number of abstract values matter globally, one needs to mine the right patterns and profile the right values for a given project. We view this result as confirmation that `SimplyTheBest` targets the right design point by providing a push-button solution to produce abstract transformers tailored to the idioms in a specific codebase.

7 Related Work

The main insight of `SimplyTheBest` is that sparsity across abstract inputs can be leveraged to improve precision in lightweight dataflow analysis. Value profiling is a closely related technique that exploits sparsity across concrete inputs to optimize code [4, 37]. There are strong connections to partial evaluation and program specialization as well [18]. We use SMT to compute precise abstract outputs following Taneja et al. [49]. This is one solution to the symbolic abstraction problem [43]; there are many others targeting more-sophisticated abstract domains [3, 23, 35, 45, 53].

Transformer Synthesis. Abstract interpretation has a long history [5, 6, 8] and there have been many attempts to systematically discover transformers for abstract domains [7, 10, 14, 26, 44, 46]. In recent years, synthesis has emerged as a promising technique to find sound and maximally precise transformers. Amurth pioneered a search technique that uses the concrete semantics of an operator to test candidate transformers, which are constructed (using Sketch [48]) by filling a template using a DSL of library functions [19]. Related works have developed the core idea along several dimensions: • handling product domains through iterative, conditional search [20, 21]; • integrating with LLVM [42] and the eBPF verifier [50], while eliminating templates through search (following STOKE [47]) and component-based program synthesis [16]; and • adapting to numerical and neural domains, where correctness depends on reasoning about numerical relations [13, 15, 40, 41]. All of the above perform synthesis for *operators*. Synthesis for instruction sequences is therefore a natural next step to improve precision. Our work takes this step in the highly practical context of static analysis for LLVM. Pasado [28] and Phocus [27] synthesize composite transformers for problems in numerical analysis. Pasado targets chain rule, product rule, and quotient rule computations for automatic differentiation. Phocus targets finite-difference schemes for nonlinear hyperbolic PDE solvers. While their core methods are specialized to the problem domain, and thus differ from ours, they share the goal of precise synthesis for instruction sequences.

Instruction Sequences. Alive2 performs translation validation for loop-free LLVM code [33]. Hydra generalizes peephole optimizations [36]. Minotaur searches for loop-free patterns to improve using SIMD instructions [29]. LPO discovers missed peephole optimizations by using an LLM to bring a degree of creativity to the table [51]. Hydride [25] uses synthesis to specialize code for modern hardware architectures; Misaal [39] improves compilation time by leveraging formal semantics of hardware instructions. These works focus on code generation rather than analysis.

8 Conclusion

Optimizing compilers should always be sound, but they don't need to optimize every program uniformly well—it is necessary, even desirable, for them to focus on generating good code for the programs that they actually compile. We created `SimplyTheBest`, which uses profiling and lightweight formal methods to synthesize precise dataflow results for the abstract values that occur in practice, when LLVM is used to optimize a large corpus of open source software. We have shown that a relatively small number of composite abstract transformers that reason about a sequence of instructions as a unit and whose implementations boil down to table lookup can increase the precision of LLVM's dataflow analysis and trigger additional program transformations.

An important next step is to ask whether compilers leverage the improved precision to generate more-efficient code—and if not, what can be done to close the gap. On a similar note, static analysis tools such as Astrée [2] might benefit from the precision gain to report fewer false positives.

A Change of Course. This paper is very different from the one we started to write. Our initial premise was: "Using LLMs + formal methods, we can synthesize composite transformers that are useful for LLVM." In terms of soundness and precision on sampled inputs, those transformers looked great on paper. However, when embedded in LLVM, they provided surprisingly few additional known bits. During the process of tracking down what we presumed was a bug in our infrastructure, we built tables of abstract inputs and noticed that they had very few entries. As the evidence mounted, it became less and less clear to us that the paper we were writing was one that needed to be written. If there is a lesson here, it is probably: *"Pay attention to what the system is telling you, and be willing to abandon your preconceptions when this is necessitated by the evidence."* Simple solutions to difficult problems are often elusive and overlooked.

Does Sparsity Generalize? We have only studied the sparsity of abstract inputs for three relatively coarse, non-relational abstract domains over integers. We chose these domains because they are load-bearing ones in LLVM, HotSpot, GCC, PyPy, and other optimizing compilers.

Why is the number of distinct abstract values seen in practice small? We hypothesize that there are two reasons. First, the abstract domains under discussion are relatively coarse and lossy—they will tend to make many different program behaviors look the same. Second, real programs do not behave arbitrarily; their behavior is typically predictable along multiple axes. For instance, one of the top-level findings in an empirical paper about value profiling by Muth et al. [37] was that variables and registers are often quasi-invariant, in the sense that *"the distribution of the values they take on is very skewed, with a small number of values occurring most of the time."*

Several open questions remain. Do our observations generalize to other non-relational integer domains? Do they generalize to non-relational domains over floating point values, pointers, or strings? What about relational abstract domains? We hope to read future work on these subjects.

The Data-Driven Future. Every optimizing compiler is data-driven, although typically the feedback cycle—which requires humans to notice a problem and then (often) different humans to fix it—is regrettably slow. The availability of a large amount of open source code is a critical enabler for the orders-of-magnitude-faster feedback loop that we get via automation. In our case,

`llvm-opt-benchmark` added even more value because it had been curated by the LLVM community. Adapting `SimplyTheBest` to a different compiler that lacks an analogous dataset (perhaps because it is for a new language) presents an interesting challenge. We conjecture that profiling abstract values in a small amount of code (or even just unit tests) would be enough to bootstrap an optimizer effort, as long as the automatic procedure is re-run frequently.

We believe that our evaluation demonstrates the utility of our approach, but we also believe that a head-to-head precision comparison against dataflow analyses in LLVM—a best-of-breed industrial compiler, the product of a massive amount of engineering—significantly under-sells the benefits of our approach. If we had brought up `SimplyTheBest` in a new compiler that lacks a mature, battle-tested dataflow framework, we expect that we would have seen a much larger increase in precision. And, indeed, we believe that workflows analogous to ours will be prevalent in the future: they not only avoid compiler bugs by construction, but also support rapid adaptation of the compiler to new circumstances, at a time scale of hours rather than months or worse.

Data Availability Statement

An artifact supporting this paper is archived at [22]. It contains `SimplyTheBest`, the Lean development described in Section 4.2, and the scripts and data needed to reproduce our results. The precision results in Table 1 and the optimization counts in Section 6 can be reproduced on a sample of `llvm-opt-benchmark` [54] in under an hour, or on the full corpus in roughly 24 hours. Two results are not reproducible from the artifact. The compile-time measurements in Table 1 require a dedicated Linux machine configured as described in Section 6, since the container cannot control core pinning, frequency scaling, or address-space randomization on the host, and the noise inside a container exceeds the effect being measured. The cross-validation in Table 2 requires rebuilding `SimplyTheBest` five times, which takes well over 72 hours.

Disclosure of Use of AI in Research

We used an LLM-based coding assistant for several tasks, primarily to assist with instrumenting LLVM. The pattern-mining hook in `ComputeKnownBits()` and `ComputeConstantRange()` was initially written by hand, but we later used an LLM-based coding assistant to extend the instrumentation to selected LLVM intrinsics. An LLM-based coding assistant also assisted with the implementation of the code that generates matching expressions in LLVM's pattern-matching DSL. These expressions are used during analysis to identify instances of mined patterns in calls to `ComputeKnownBits()` and `ComputeConstantRange()`. The implementation was designed to produce match statements that follow the conventions of LLVM's existing pattern-matching infrastructure. The code used to generate tables for the evaluation was likewise written with assistance from an LLM-based coding assistant. For the part of the evaluation where we instrumented LLVM optimizations driven by known bits, the initial logging facility was drafted by a coding assistant. The initial instrumentation was performed by hand, but then we asked an LLM to do a code review, and it pointed out several missed logging opportunities. We then asked the LLM to go ahead and fill in the gaps that it noticed. We reviewed and stand behind all of the LLM-assisted code. The Lean development described in Section 4.2 was also LLM-assisted. We first developed completeness proofs by hand on paper, and then used Codex to mechanize the proofs in Lean. We verified that the formal statements match the definitions and claims in the paper. The assistant found several bugs in our paper proofs, for which we are grateful.

Acknowledgments

The project is supported in part by a Microsoft Faculty Fellowship; a UCSD JSOE Scholarship; Schmidt Foundation; and NSF under grants CCF-2422214, CCF-2506134, CCF-2446711, and SSF-2546821. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities. Loris D'Antoni holds concurrent appointments as a Professor at the University of California San Diego and as a Code Metal Scholar. This paper describes work performed at the University of California San Diego and is not associated with Code Metal. We thank the POPL reviewers for their detailed feedback. Special thanks to Marion (age 2) for entertainment during several zoom calls.

References

[1] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *FREENIX Track*. USENIX Association, 41–46.

[2] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *PLDI*. ACM, 196–207. doi:10.1145/781131.781153

[3] Jörg Brauer and Andy King. 2010. Automatic Abstraction for Intervals Using Boolean Formulae. In *SAS*. Springer, 167–183. doi:10.1007/978-3-642-15769-1_11

[4] Brad Calder, Peter Feller, and Alan Eustace. 1999. Value Profiling and Optimization. *Journal of Instruction Level Parallelism* 1 (1999). http://www.jilp.org/vol1/v1paper2.pdf

[5] Patrick Cousot. 2024. A Personal Historical Perspective on Abstract Interpretation. In *The French School of Programming*. Springer, 205–239. doi:10.1007/978-3-031-34518-0_9

[6] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252. doi:10.1145/512950.512973

[7] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. ACM, 269–282. doi:10.1145/567752.567778

[8] Patrick Cousot and Radhia Cousot. 2014. Abstract interpretation: past, present and future. In *CSL / LICS*. ACM, 2:1–2:10. doi:10.1145/2603088.2603165

[9] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTRÉE analyzer. In *ESOP*. Springer, 21 pages. doi:10.1007/b107380

[10] Matt Elder, Junghee Lim, Tushar Sharma, Tycho Andersen, and Thomas Reps. 2014. Abstract Domains of Affine Relations. *ACM Transactions on Programming Languages and Systems* 36, 4, Article 11 (2014), 73 pages. doi:10.1145/2651361

[11] Roberto Giacobazzi and Elisa Quintarelli. 2001. Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking. In *SAS*. Springer Berlin Heidelberg, 356–373. doi:10.1007/3-540-47764-0_20

[12] Roberto Giacobazzi and Francesco Ranzato. 2025. The Best of Abstract Interpretations. *Proc. ACM Program. Lang.* 9, POPL, Article 46 (2025). doi:10.1145/3704882

[13] Shaurya Gomber, Debangshu Banerjee, and Gagandeep Singh. 2026. Evolving Abstract Transformers for Gradient-Guided, Adaptable Abstract Interpretation. *Proceedings of the ACM on Programming Languages* 10, PLDI, Article 268 (2026), 25 pages. doi:10.1145/3808346

[14] Susanne Graf and Hassen Saidi. 1997. Construction of abstract state graphs with PVS. In *CAV*. Springer, 72–83. doi:10.1007/3-540-63166-6_10

[15] Qiuhan Gu, Avaljot Singh, and Gagandeep Singh. 2026. SAIL: Sound Abstract Interpreters with LLMs. *Proceedings of the ACM on Programming Languages* 10, PLDI, Article 230 (2026), 26 pages. doi:10.1145/3808308

[16] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. In *PLDI*. ACM, 62–73. ISBN 9781450306638. doi:10.1145/1993498.1993506

[17] Jaromir Hamala. 2026. The mask that compiles to nothing: how HotSpot's JIT learned to reason about bits. https://questdb.com/blog/jvm-jit-known-bits/ Accessed 2026-07-03.

[18] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall. ISBN 978-0-13-020249-9.

[19] Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas W. Reps, and Subhajit Roy. 2022. Synthesizing Abstract Transformers. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1291–1319. doi:10.1145/3563334

[20] Pankaj Kumar Kalita, Thomas Reps, and Subhajit Roy. 2024. Synthesizing Abstract Transformers for Reduced-Product Domains. In *SAS*. Springer-Verlag, 147–172. doi:10.1007/978-3-031-74776-2_6

[21] Pankaj Kumar Kalita, Thomas Reps, and Subhajit Roy. 2026. Automated Abstract Transformer Synthesis for Reduced Product Domains. *ACM Transactions on Software Engineering Methodology* 35, 2, Article 48 (2026), 41 pages. doi:10.

1145/3733716

[22] Dominic Kennedy, Xuanyu Peng, Yuyou Fan, John Regehr, Loris D'Antoni, and Ben Greenman. 2026. Artifact for "A Fistful of Abstract Values: Crafting the Transformers Compilers Actually Need". doi:10.6084/m9.figshare.34415610.v1 Software artifact.

[23] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas W. Reps. 2018. Non-linear Reasoning for Invariant Synthesis. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 54:1–54:33. doi:10.1145/3158142

[24] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (2015), 573–609. doi:10.1007/s00165-014-0326-7

[25] Akash Kothari, Abdul Rafae Noor, Muchen Xu, Hassam Uddin, Dhruv Baronia, Stefanos Baziotis, Vikram Adve, Charith Mendis, and Sudipta Sengupta. 2024. Hydride: A Retargetable and Extensible Synthesis-based Compiler for Modern Hardware Architectures. In *ASPLOS*. ACM, 514–529. doi:10.1145/3620665.3640385

[26] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. 2006. SMT Techniques for Fast Predicate Abstraction. In *CAV*. Springer, 424–437. doi:10.1007/11817963_39

[27] Jacob Laurel, Ignacio Laguna, and Jan Hückelheim. 2025. Synthesizing Sound and Precise Abstract Transformers for Nonlinear Hyperbolic PDE Solvers. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 1063–1091. doi:10.1145/3763088

[28] Jacob Laurel, Siyuan Brant Qian, Gagandeep Singh, and Sasa Misailovic. 2023. Synthesizing Precise Static Analyzers for Automatic Differentiation. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 1964–1992. doi:10.1145/3622867

[29] Zhengyang Liu, Stefan Mada, and John Regehr. 2024. Minotaur: A SIMD-Oriented Synthesizing Superoptimizer. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2, Article 326 (2024), 25 pages. doi:10.1145/3689766

[30] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. doi:10.1145/3428264

[31] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* 7, PLDI, Article 181 (June 2023), 22 pages. doi:10.1145/3591295

[32] LLVM. 2026. Benchmarking Tips. https://llvm.org/docs/Benchmarking.html

[33] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *PLDI*. ACM, 65–79. doi:10.1145/3453483.3454030

[34] MathWorks. 2026. Polyspace: Test software and assess code quality. https://www.mathworks.com/products/polyspace.html Accessed 2026-02-25.

[35] David Monniaux. 2010. Quantifier Elimination by Lazy Model Enumeration. In *CAV*. Springer, 585–599. doi:10.1007/978-3-642-14295-6_51

[36] Manasij Mukherjee and John Regehr. 2024. Hydra: Generalizing Peephole Optimizations with Program Synthesis. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1, Article 120 (2024), 29 pages. doi:10.1145/3649837

[37] Robert Muth, Scott A. Watterson, and Saumya K. Debray. 2000. Code Specialization Based on Value Profiles. In *SAS*. Springer-Verlag, 340–359. doi:10.1007/978-3-540-45099-3_18

[38] Aina Niemetz and Mathias Preiner. 2023. Bitwuzla. In *CAV*. Springer, 3–17. doi:10.1007/978-3-031-37703-7_1

[39] Abdul Rafae Noor, Dhruv Baronia, Akash Kothari, Muchen Xu, Charith Mendis, and Vikram S. Adve. 2025. MISAAL: Synthesis-Based Automatic Generation of Efficient and Retargetable Semantics-Driven Optimizations. *Proceedings of the ACM on Programming Languages* 9, PLDI, Article 198 (2025), 24 pages. doi:10.1145/3729301

[40] Brandon Paulsen and Chao Wang. 2022. Example Guided Synthesis of Linear Approximations for Neural Network Verification. In *CAV*. Springer, 149–170. doi:10.1007/978-3-031-13185-1_8

[41] Brandon Paulsen and Chao Wang. 2022. LinSyn: Synthesizing Tight Linear Bounds for Arbitrary Neural Network Activation Functions. In *TACAS*. Springer, 357–376. doi:10.1007/978-3-030-99524-9_19

[42] Xuanyu Peng, Dominic Kennedy, Yuyou Fan, Ben Greenman, John Regehr, and Loris D'Antoni. 2026. Nice to Meet You: Synthesizing Practical MLIR Abstract Transformers. *Proceedings of the ACM on Programming Languages* 10, POPL, Article 80 (2026), 29 pages. doi:10.1145/3776722

[43] Thomas Reps, Mooly Sagiv, and Greta Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *VMCAI*. Springer, 252–266. doi:10.1007/978-3-540-24622-0_21

[44] Thomas W. Reps and Aditya V. Thakur. 2016. Automating Abstract Interpretation. In *VMCAI*. Springer, 3–40. doi:10.1007/978-3-662-49122-5_1

[45] Fabian Ritter. 2015. *Compiler Optimizations using Symbolic Abstraction.* Bachelor's Thesis. Saarland University. https://compilers.cs.uni-saarland.de/publications/theses/ritter_bsc.pdf

[46] Erika Rice Scherpelz, Sorin Lerner, and Craig Chambers. 2007. Automatic inference of optimizer flow functions from semantic meanings. In *PLDI*. ACM, 135–145. doi:10.1145/1250734.1250750

[47] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *ASPLOS*. ACM, 305–316. doi:10.1145/2451116.2451150

[48] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5-6 (2013), 475–495. doi:10.1007/s10009-012-0249-7

[49] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing static analyses for precision and soundness. In *CGO*. ACM, 81–93. doi:10.1145/3368826.3377927

[50] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2025. Automatic Synthesis of Abstract Operators for eBPF. In *Workshop on eBPF and Kernel Extensions*. ACM, 15–21. doi:10.1145/3748355.3748361

[51] Zhenyang Xu, Hongxu Xu, Yongqiang Tian, Xintong Zhou, and Chengnian Sun. 2026. LPO: Discovering Missed Peephole Optimizations with Large Language Models. In *ASPLOS*. ACM, 1136–1150. doi:10.1145/3779212.3790184

[52] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. *SIGPLAN Not.* 46, 6 (June 2011), 283–294. doi:10.1145/1993316.1993532

[53] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2021. Program Analysis via Efficient Symbolic Abstraction. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–32. doi:10.1145/3485495

[54] Yingwei Zheng. 2023. LLVM Opt Benchmark. https://github.com/dtcxzyw/llvm-opt-benchmark Archived on 2026-03-17.

A Completeness in LLVM Domains

This section demonstrates how the abstract definitions of forward and backward completeness are realized in the `KnownBits`, `UConstRange` and `SConstRange` domains. Each subsection presents formal definitions and examples of complete and incomplete operators.

Some LLVM operations produce poison on certain inputs; for example, `add nuw` produces poison when the sum overflows. LLVM's analyses are poison-oblivious, so we model a poisoned input as producing no result. The criteria below are stated for total operations. Each subsection notes how poison-producing operations are handled.

A.1 Completeness Criteria for `KnownBits`

An operation is forward complete in `KnownBits` iff each bit of the m-bit output is determined by a disjoint set of bits of the n-bit input. Uncertainty in one input bit can affect at most one output bit, so tracking each bit independently loses no correlations. Formally, for a unary operation, all pairs of output-bit dependency sets must be disjoint:

$$\forall\, i, j \in [m].\ i \neq j \implies \mathrm{BitDep}_{op}(i) \cap \mathrm{BitDep}_{op}(j) = \emptyset.$$

Here, the dependency set of output bit i contains the input bits that can affect it:

$$\mathrm{BitDep}_{op}(i) := \left\{ j \in [n] \;\middle|\; \exists x \in C.\ op(x)[i] \neq op(x \ \mathtt{xor}\ 2^j)[i] \right\}.$$

When an operation has two or more inputs, the same condition is applied separately to each operand. For an operation with inputs x and y, the dependency set of output bit i within x is

$$\mathrm{BitDep}^{x}_{op}(i) := \left\{ j \in [n] \;\middle|\; \exists\, x, y \in C.\ op(x, y)[i] \neq op(x \ \mathtt{xor}\ 2^j,\ y)[i] \right\},$$

and BitDep^{y}_{op} is defined symmetrically. For example, for a binary operation over inputs x and y, the dependency sets $\mathrm{BitDep}^{x}_{op}(\cdot)$ must be pairwise disjoint, and the dependency sets $\mathrm{BitDep}^{y}_{op}(\cdot)$ must be pairwise disjoint.

A `KnownBits` operation is backward complete when every output bit is determined by at most one bit of each input operand. Each output bit can then be traced back to a single bit per input, so abstraction by `KnownBits` captures all information the operation can observe. Formally, for a unary operation, each output bit i must be computable by a Boolean function ϕ_i applied to one input bit:

$$\forall\, i \in [m].\ \exists\, \pi_i \in [n],\ \phi_i : \{0,1\} \to \{0,1\}.\ \forall\, x \in C.\ op(x)[i] = \phi_i(x[\pi_i]).$$

For higher-arity operations, the condition applies to each operand with the other operands held fixed. For every output bit i, every operand, and every fixed value of the remaining operands, $op[i]$ must be computable from a single bit of that operand, and the chosen bit may depend on the values of the remaining operands. For example, if op takes inputs x and y, then for each fixed y there must be an index $\pi^x_i(y)$ and a Boolean function ϕ with $op(x, y)[i] = \phi(x[\pi^x_i(y)])$ for all x, and symmetrically for y with x fixed. The dependency criteria assume a total operation, since they only check how each output bit depends on the input bits, but for a poison-producing operation the input bits also determine whether there is a result at all, and this is a dependence the dependency sets cannot express. For such operations we characterize completeness over the input-output sets, where the non-existence of a result is yet another observable fact.

Examples.

- `xor` is forward complete and backward complete because each output bit depends on exactly one bit from each input.

- `add` is neither forward complete nor backward complete because of carries. Forward completeness fails because a low bit can influence several higher bits. Backward completeness fails because an output bit can depend on many lower input bits.
- `or disjoint` is forward complete but not backward complete. On the inputs where it is defined, it has the same per-bit dependencies as `or`, which is complete in both directions. Backward completeness fails because a poison-producing input contributes no result. The defined members of an input set may share a known bit that the defined members of its closure do not, so the abstract output computed from the closure is less precise than the abstract output computed from the set itself.
- `sext` is backward complete but not forward complete. Forward completeness fails because the sign bit of the input is copied to multiple positions in the output. Backward completeness succeeds because each output bit depends on only one input bit.

A.2 Completeness Criteria for `UConstRange` and `SConstRange`

The signed and unsigned variants of `ConstantRange` use the same completeness criteria. The only difference is the order used to compare values: `UConstRange` uses the unsigned order, while `SConstRange` uses the signed order.

An operation is forward complete for an interval domain if mapping it over an interval produces a set with no gaps—that is, another interval. For a unary operation, this means that for every interval $[lo, hi]$,

$$\{\, op(i) \mid i \in [lo, hi] \,\} = \left[\min_{i \in [lo,hi]} op(i),\ \max_{i \in [lo,hi]} op(i) \right].$$

For higher-arity operations, the same condition is applied to the Cartesian product of the input intervals. Thus the set of possible outputs must contain every value between its minimum and maximum in the appropriate signed or unsigned order. These criteria are stated over the results at the endpoints, so they apply unchanged to poison-producing operations, which have no result.

A `ConstantRange` operation is backward complete when the abstract output is determined by the endpoints of the input interval. For a unary operation, this means that for every interval $[lo, hi]$,

$$\min_{i \in [lo,hi]} op(i) = \min\{op(lo), op(hi)\}, \qquad \max_{i \in [lo,hi]} op(i) = \max\{op(lo), op(hi)\}.$$

For higher-arity operations, the analogous condition is that the output bounds are determined by the corners of the input box. In that case, closing an intermediate set to an interval cannot change the interval computed by the consumer.

Examples.

- `UsubSat` subtracts two unsigned numbers and clamps its minimum output to zero. It is both forward complete and backward complete in `UConstRange`.
- `UmulSat` multiplies two unsigned n-bit numbers and clamps its output to the interval $[0, 2^n)$. It is backward complete in `UConstRange` because the output interval is determined by endpoint values. It is not forward complete because the image of an interval may contain gaps. For example, mapping `UmulSat` over the intervals $[1, 2]$ and $[1, 2]$ produces $\{1, 2, 4\}$, which is not an interval.
- `abs` is forward complete but not backward complete in `UConstRange`. Backward completeness fails on the interval $[127, 129]$ at width 8: both endpoints map to 127, but the interior value 128 maps to 128, so the upper endpoint of the output occurs at an interior value.
- `xor` is neither forward complete nor backward complete for interval domains. For example, over the intervals `x` $= [1, 3]$ and `y` $= [2, 2]$, the output set has gaps and its bounds are not determined by endpoint reasoning alone.

Table 3. DSL operations, their definitions in LLVM IR, and abstract transformer completeness, Part 1/2.

		KB		UCR		SCR	
Operation	**Definition in LLVM IR**	→	←	→	←	→	←
Abs	`@llvm.abs.iN(iN %x, i1 false)`			✓			
AbsUndef	`@llvm.abs.iN(iN %x, i1 true)`			✓		✓	
PopCount	`@llvm.ctpop.iN(iN %x)`						
CountLZero	`@llvm.ctlz.iN(iN %x, i1 false)`			✓	✓		
CountLZeroUndef	`@llvm.ctlz.iN(iN %x, i1 true)`			✓			
CountRZero	`@llvm.cttz.iN(iN %x, i1 false)`						
CountRZeroUndef	`@llvm.cttz.iN(iN %x, i1 true)`						
Add	`add iN %x, %y`						
AddNsw	`add nsw iN %x, %y`					✓	
AddNuw	`add nuw iN %x, %y`			✓			
AddNswNuw	`add nsw nuw iN %x, %y`						
And	`and iN %x, %y`	✓	✓				
Ashr	`ashr iN %x, %y`						
AshrExact	`ashr exact iN %x, %y`						
Lshr	`lshr iN %x, %y`						
LshrExact	`lshr exact iN %x, %y`						
Mul	`mul iN %x, %y`						
MulNsw	`mul nsw iN %x, %y`						
MulNuw	`mul nuw iN %x, %y`						
MulNswNuw	`mul nsw nuw iN %x, %y`						
Or	`or iN %x, %y`	✓	✓				
OrDisjoint	`or disjoint iN %x, %y`	✓					
Sdiv	`sdiv iN %x, %y`						
SdivExact	`sdiv exact iN %x, %y`						

B Supported Operators

Tables 3 and 4 list the operations in the pattern DSL that `SimplyTheBest` supports. Alongside each operation, we give a semantically equivalent definition in primitive LLVM IR. Finally, they note whether each DSL operation is forward complete (→) or backward complete (←) in each of the three domains. Each classification of completeness (or lack thereof) is formalized in Lean.

C Experiment Details

The following command makes a release build of LLVM:

```
cmake -S llvm -B build-release -G Ninja \
  -DCMAKE_BUILD_TYPE=Release            \
  -DLLVM_TARGETS_TO_BUILD="X86"        \
  -DLLVM_USE_LINKER=lld                \
  -DLLVM_ENABLE_ASSERTIONS=OFF
```

We note the following departures from the best practices recommended by the LLVM benchmarking guide [32]. We do not use static linking because it requires too much manual instrumentation. We cannot fit the entire `llvm-opt-benchmark` on an in-memory filesystem, but we do store the data on an SSD to speed up access. We use `cpusets` instead of `cset shield` because the former is recommended by our OS, Ubuntu 26.04.

Received 2026-07-09; accepted 2026-10-05

Table 4. DSL operations, their definitions in LLVM IR, and abstract transformer completeness, Part 2/2.

		KB		UCR		SCR	
Operation	**Definition in LLVM IR**	→	←	→	←	→	←
Shl	`shl iN %x, %y`						
ShlNsw	`shl nsw iN %x, %y`						
ShlNuw	`shl nuw iN %x, %y`						
ShlNswNuw	`shl nsw nuw iN %x, %y`						
Srem	`srem iN %x, %y`						
Sub	`sub iN %x, %y`						
SubNsw	`sub nsw iN %x, %y`					✓	
SubNuw	`sub nuw iN %x, %y`			✓			
SubNswNuw	`sub nsw nuw iN %x, %y`						
Udiv	`udiv iN %x, %y`						
UdivExact	`udiv exact iN %x, %y`						
Urem	`urem iN %x, %y`						
Xor	`xor iN %x, %y`	✓	✓				
Umax	`@llvm.umax.iN(iN %x, iN %y)`			✓	✓		
Umin	`@llvm.umin.iN(iN %x, iN %y)`			✓	✓		
Smax	`@llvm.smax.iN(iN %x, iN %y)`					✓	✓
Smin	`@llvm.smin.iN(iN %x, iN %y)`					✓	✓
SaddSat	`@llvm.sadd.sat.iN(iN %x, iN %y)`					✓	✓
UaddSat	`@llvm.uadd.sat.iN(iN %x, iN %y)`			✓	✓		
SsubSat	`@llvm.ssub.sat.iN(iN %x, iN %y)`					✓	✓
UsubSat	`@llvm.usub.sat.iN(iN %x, iN %y)`			✓	✓		
SmulSat	`@llvm.smul.fix.sat.iN(iN %x, iN %y, i32 0)`						✓
UmulSat	`@llvm.umul.fix.sat.iN(iN %x, iN %y, i32 0)`				✓		
SshlSat	`@llvm.sshl.sat.iN(iN %x, iN %y)`						
UshlSat	`@llvm.ushl.sat.iN(iN %x, iN %y)`						
ICmpEq	`icmp eq iN %x, %y`	✓		✓		✓	
ICmpNe	`icmp ne iN %x, %y`	✓		✓		✓	
ICmpSlt	`icmp slt iN %x, %y`	✓		✓		✓	✓
ICmpSle	`icmp sle iN %x, %y`	✓		✓		✓	✓
ICmpSgt	`icmp sgt iN %x, %y`	✓		✓		✓	✓
ICmpSge	`icmp sge iN %x, %y`	✓		✓		✓	✓
ICmpUlt	`icmp ult iN %x, %y`	✓		✓	✓	✓	
ICmpUle	`icmp ule iN %x, %y`	✓		✓	✓	✓	
ICmpUgt	`icmp ugt iN %x, %y`	✓		✓	✓	✓	
ICmpUge	`icmp uge iN %x, %y`	✓		✓	✓	✓	
Trunc	`trunc iN %x to i1`	✓	✓	✓		✓	
Zext	`zext i1 %x to iN`	✓	✓	✓	✓	✓	✓
Sext	`sext i1 %x to iN`		✓		✓	✓	✓
Select	`select i1 %cond, iN %x, iN %y`		✓		✓		✓