Unbounded Superoptimization

Abhinav Jangda
Indian Institute of Technology (BHU), India

Greta Yorsh
Queen Mary University of London, UK

Abstract

Our aim is to enable software to take full advantage of the capabilities of emerging microprocessor designs without modifying the compiler.

Towards this end, we propose a new approach to code generation and optimization. Our approach uses an SMT solver in a novel way to generate efficient code for modern architectures and guarantee that the generated code correctly implements the source code. The distinguishing characteristic of our approach is that the size of the constraints does not depend on the candidate sequence of instructions.

To study the feasibility of our approach, we implemented a preliminary prototype, which takes as input LLVM IR code and uses Z3 SMT solver to generate ARMv7-A assembly. The prototype handles arbitrary loop-free code (not only basic blocks) as input and output. We applied it to small but tricky examples used as standard benchmarks for other superoptimization and synthesis tools. We are encouraged to see that Z3 successfully solved complex constraints that arise from our approach.

This work paves the way to employing recent advances in SMT solvers and has a potential to advance SMT solvers further by providing a new category of challenging benchmarks that come from an industrial application domain.

CCS Concepts • **Software and its engineering → Compilers**; • **Theory of computation** → *Logic and verification*;

Keywords superoptimization, software synthesis, code generation and optimization, SMT solver, constraint solvers, first order logic, instruction set architecture

ACM Reference Format:
Abhinav Jangda and Greta Yorsh. 2017. Unbounded Superoptimization. In *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3133850.3133856

Onward!'17, October 25–27, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5530-8/17/10…$15.00
https://doi.org/10.1145/3133850.3133856

1 Introduction

Production compilers such as GCC, LLVM, Intel C compiler, and Microsoft Visual C compiler can generate efficient code for popular target hardware. Ideally, advances in hardware design would directly translate to performance improvements in software. In reality, this involves a manual process of tuning a sophisticated production compiler or hardware-specific rewriting of code. This process is challenging even for the few experts who possess the required range of skills. Moreover, any errors introduced in this process affect the entire software stack and might compromise its reliability and security.

The growing variety and complexity of hardware designs, spanning the spectrum of low-power and high-performance computing, makes it even more challenging for software development to keep up. Practitioners argue that general-purpose optimizing compilers are falling further behind the actual capabilities of modern processors. [1]

The aim of our work is to enable software to take full advantage of the capabilities of emerging microprocessor designs without modifying the compiler.

Towards this end, we propose a new approach to code generation and optimization. It uses constraint solving in a novel way to generate efficient code and guarantee that the generated code correctly implements the source code.

We define algorithms parametric in the formal specification of application level instruction semantics (including control, data access in registers and memory, bitwise operations, vector operations) and a separate cost model of these instructions. Varying the parameters allows us to (i) target new microarchitectures without changing the compiler, and (ii) develop cost models with different levels of precision, for example proprietary models can be more accurate.

Our work is inspired by recent advances in Satisfiability Modulo Theories (SMT) and the growing availability of formal specifications for Instruction Set Architectures (ISAs).

Satisfiability Modulo Theories refers to the following decision problem: given a first order logic formula, is the formula satisfiable with respect to a combination of background theories of classical first-order logic with equality. SMT can also be thought of as a form of constraint satisfaction problem. The problem is decidable for interesting fragments of first order logic. Modern SMT solvers such as Z3 [11] and CVC4 [3] are very efficient in handling known decidable fragments. Moreover, these solvers use heuristics

[1] ETAPS 2015 Inivited Tutorial titled "The death of optimizing compilers" by Daniel J. Bernstein.

and user-defined strategies to handle arbitrary first order formulas. If an SMT solver proves that a formula is satisfiable, the solver can also produce (a finite representation of) a model of the formula.

Our approach relies on the theory of bit-vectors in combination with extensional arrays and the standard equality and uninterpreted functions, which are decidable for quantifier free formulas. The novelty of our approach lies in the judicious use of quantifiers.

Superoptimization and Synthesis using SMT Solvers Our approach is closely-related to superoptimization and synthesis [19, 20, 29, 35, 40, 42]. These methods search through the space of candidate instruction sequences of increasing length and use SAT or SMT solvers to check whether a candidate correctly implements the source code. This works well for optimizing short straight-line code. With the increase in length of the target sequence, the search space dramatically increases, posing a challenge for existing approaches.

Our approach to code generation and optimization uses an SMT solver in a novel way. The distinguishing characteristic of our approach is that the size of the constraints does not depend on the candidate instruction sequence.

We believe that this is key to scalability, in particular when going beyond superoptimization of straightline code: a solver has the potential to reuse more of its reasoning during its search and incorporate cost models into the search. It also provides a way to support cost models that accurately reflect modern microarchitecture features (such as multiple issue, different register banks, different decode speeds, memory latency, etc), whereas for standard superoptimization and synthesis the cost is usually the length of the generated sequence.

Our encoding is closely related to Denali [21] in that it expresses all target instruction sequences of any length that correctly implement the source code. The authors of Denali make the following observation about using such an encoding, in the early days of SMT solvers: *"Conjectures of this form are daunting for two reasons: First, the universal quantifier nested within the existential quantifier is difficult for automatic theorem provers to handle. Second, the many cases in the definitions [of ISA semantics] tend to lead automatic theorem provers into a morass of case analyses."*

To cope with it, Denali implemented a specialized solver based on equality-preserving transformations. These transformations were designed by experts and implemented using matching on E-graphs. Today, E-graphs matching is tightly integrated within modern SMT solvers [12]. Moreover, recent advances in SMT solvers include solving optimization problems [6, 7], strategies to guide heuristic search performed by highly-optimized SMT solvers [13] quantifiers [5, 34], bit-vectors [16, 49], and model generation [48].

These advances in SMT solvers provide an unprecedented opportunity for optimizing compilers. Nevertheless, all of the recent work on superoptimization and synthesis still avoids quantifier alternation in the encoding they pass to SMT solvers. We set about to reexamine this design choice.

Shifting the Search into the Solver Instead of searching through the space of candidate instruction sequences and then calling the solver on some of them, our approach is to guide the search performed within an SMT solver. SMT solvers are designed to prune incorrect solutions from the search space. We conjecture that tighter integration with an SMT solver will provide us with an efficient way to search through the much smaller space of correct solutions towards finding an optimal one.

For this idea to work, our encoding in combination with solver strategies should capture sufficient information about structure of the source program and the target architecture. Then, a variety of application domain specific techniques can be used within an SMT solver to speed up the search.

For example, data-driven stochastic methods related to those used for superoptimization [35, 37] can also be useful within an SMT solver to speed up the search, e.g, [16]. Another example is divide-and-conquer strategies used in synthesis [42] to break the formula into independent subformulas. SMT solvers perform this kind of operations and may be able to make a better-informed decision about it.

The first step will be to explore existing algorithms and tactics exposed by SMT solvers for guiding the search. Then, we expect that some changes to solver's implementation may be needed to speed up the search and possibly to improve quantifier instantiation for our application domain.

Developing methods through which users of SMT solvers can control the heuristic search was posed as a research challenge in SMT community [13]. Our encoding can serve as a source of benchmarks to drive changes in solvers, both in terms of algorithms and tactics exposed to users. As the solvers evolve, there may be better ways to express the constraints, for example using different combination of theories or tactics. Our infrastructure enables us to experiment with different ways of expressing the constraints.

Our encoding can be used in a range of contexts including traditional superoptimization, code generation, binary translation, binary synthesis, binary optimization, peephole generation, and possibly even dynamic optimization.

Compilation Time Traditionally, stringent requirements on compilation time dictate compromises on the quality of generated code. Modern optimizing compilers apply particular sequences of transformation, each of which makes heuristic decisions about performance of some aspect of generated code. Decisions are often made without regard to dependencies between transformations. For example, there are dependencies between register allocation, instruction selection, instruction scheduling, and peephole optimizations. Such decisions are necessarily suboptimal and easy to get wrong. The space of possible transformations and

their ordering has been the subject of extensive research. Recently, Milepost [17] used machine learning to find best combination.

Superoptimization, pioneered by [29] avoids, to some extent, the problem of ordering transformations, and eliminates the need for peepholes optimizations. In some cases, it can make optimal decisions for the combination of register allocation, instruction selection and instruction scheduling. Superoptimization approach can generate code that cannot be produced by existing compilers or methods such as [17].

Superoptimization is considered too slow to use during software development, but may be acceptable in some situations, such as optimizing the body of hot loops and compute kernels for a release version, or at compiler development time. In [2, 8], superoptimization is used to generate a library of peephole transformations from ISA specifications. These tools need to rerun whenever ISA or cost model changes and might produce too many peephole transformations that might slow down compilation. To address it, Alive [28] involves compiler developers in defining peepholes that deemed to be beneficial.

Our approach provides the programmer with a fine-grained control on how much time is spent in compilation vs the quality of generated code. If a programmer increases the time budget allotted for compilation, our approach will generate increasingly better code in terms of running time, code size, power consumption, energy efficiency, etc. With our approach, code generation and optimization can stop at any time with a correct (but possibly suboptimal) solution, as explained in Section 2. Moreover, existing solution can be improved upon later on when budget increased. In contrast, traditional superoptimization cannot return a correct solution until the end.

Correctness Guarantees Our approach provides correctness guarantees per compilation, in the sense of translation validation [4, 31, 43]. An alternative approach is to prove correctness of the compiler itself, e.g., [50]. Recently, CompCert [25] and CakeML [44] have been able to certify their compiler backends that implement sophisticated optimization, getting closer to the quality of code generated by production compilers. It would be interesting to apply these techniques to certify a superoptimizer.

Formal Specification of ISAs There are several versions of accurate, detailed, and nearly-complete ISA specifications for common architectures. Hardware design vendors have proprietary ISA specifications. Notably, ARM's official formal ISA specification for ARMv8-A, described in [33], has just been released [1]. Other open-source versions originated from the research community, such as [14, 15, 18, 26, 27]. They are expressed using domain specific languages, some of which can be automatically translated to higher order logic for use in mechanized software verification with interactive theorem provers such as Coq, HOL, Isabelle, and ACL2. We are not aware of a specification language that provides an interface to first order logic or SMT solvers as required for our approach.

Preliminary Prototype We implemented a preliminary prototype of our approach, using the LLVM compiler and the Z3 SMT solver, targeting ARM. The prototype uses a small subset of ARMv7-A, sufficient by design for our microbenchmarks so far. We are in the process of integrating ARM's official formal ISA specification for ARMv8-A [1].

The prototype supports arbitrary loop-free code (not only basic blocks) as input and output. Most of our benchmarks are straightline code.

The prototype successfully handles some small but tricky examples used as standard benchmarks for other superoptimization and synthesis tools. The quality of code generated by our prototype compares favorably with production compilers and other superoptimization and synthesis tools. We have not compared performance because recent superoptimization tools [35, 42] target IA-32 and x86-64 architectures. To the best of our knowledge, ours is the first application of superoptimization to a modern ARM architecture.

1.1 Contributions of this Paper

- The idea of shifting the search for optimal instruction sequences into the SMT solver. This provides a novel way of structuring the search.
- A first-order encoding that realizes this idea.
- A preliminary prototype that enables us to experiment with different ways to express the encoding using SMT solvers. This provides the infrastructure for the feasibility study reported in this paper, and further experiments.
- A feasibility study that provides an early indication of the viability of our approach.

Novelty of the Encoding The key novelty is that we encode the semantics of all instruction sequences (of unbounded length), rather than encoding the semantics of some candidate instruction sequences of a given length. This is achieved using more quantifiers than existing approaches.

An additional novel aspect of our encoding is that it goes beyond straight-line code: it can handle loop-free code as input and output. The simulation relation is also inferred by the solver. This is achieved by combining several techniques inspired by existing work, as discussed in Section 3.

New Benchmarks for SMT Solvers This work paves the way to employing recent advances in SMT solvers and has a potential to advance SMT solvers further by providing a new category of challenging benchmarks that people care about. That is, benchmarks that come from an industrial application domain, not randomly generated.

Our prototype generates constraints that combine quantifiers, bit-vectors, and arrays. Even if currently SMT solvers cannot handle these constraints very well, they can serve as

benchmarks. We hope they can be included in Satisfiability Modulo Theories Competition (SMT-COMP), known as a strong driver of research and performance improvements of SMT solvers.

Feasibility Study In Section 4, we report on experiments we performed to find out whether an SMT solver is at all capable of solving the complex constraints that arise from our approach. The key technical challenge has to do with quantifiers present in our encoding, but not in other methods. We are encouraged that Z3 solved them, but it took a long time. We do not have evidence yet that our approach scales better than existing approaches or generates better code.

Scope of the Experiments There are many important aspects that we have not explored yet, including:

- tactics for controlling the search from within Z3,
- cost functions other than the standard one, which is length of the instruction sequence,
- optimization queries using MaxSMT,
- handling memory operations, which results in one quantifier alternation in the constraints.

2 Overview

The problem we are addressing can be stated as follows. Given code p, generate code s such that

- **(Correctness)** s implements p, i.e., the observable behaviors of s are a subset of the observable behaviors of p.
- **(Optimality)** the cost of s is minimal with respect to cost function c, i.e., $c(s) = min\{c(s') \mid s' \text{ implements } p\}$

This problem statement is parametric in the source and target languages as well as the cost function of the output code.

This problem arises in several contexts with different source and target languages. In the context of an optimizing compiler and superoptimization, both the input p and the output s code is expressed in compiler's intermediate representation, such as LLVM IR or GCC RTL. In the context of code generation, the input p is an intermediate representation and the output s is assembly (or machine code) of the target architecture. For binary translators, both the input p and the output s are in assembly (or machine code) corresponding to the source architecture and target architecture, respectively.

Synthesis does not fit with the problem statement above, because synthesis takes as input a first-order logic formula, rather than code p, as explained in [42]. Our approach can also be used in the context of synthesis, because the first step of our approach is to generate a logical formula that corresponds to p in a target-independent way. We can skip this step in the context of synthesis and use the input formula directly in place of the logical formula for p.

In our prototype, the input p is expressed in LLVM IR and the output s is generated as ARMv7-A assembly.

Algorithm 1 Unbounded Superoptimization (novel)

Parameters: target architecture specification a, cost function c
Input: code p
Output: code s that implements p in a with minimal cost $c(s)$, or FAIL

```
1: function UnboundedSuperoptimizer(p, a, c)
2:     χ ← encodeCorrectness(p, a)
3:     if not satisfiable(χ) then return FAIL
4:     repeat
5:         m ← getModel(χ)
6:         χ ← χ ∧ encodeBound(m, c)
7:     until not satisfiable(χ)
8:     s ← getCode(m)
9:     return s
```

2.1 Algorithm

A pseudocode of unbounded superoptimization is shown in Algorithm 1. It is parametric in the semantics of the target architecture a and the cost function c. It takes as input code p and returns code s. In Line 2, encodeCorrectness, parametric in a, takes p and returns a formula χ, such that if χ is satisfiable, then there exists a target instruction sequence that correctly implements the source code. Note that encodeCorrectness captures the correctness requirement above, but does not place any requirements on the cost of the generated sequence. See details of encodeCorrectness in Section 2.2.

Line 3 checks satisfiability of χ to ensure existence of a model. The loop in Lines 4-7 searches for a model with the minimal cost. In Line 5, getModel returns a model m of χ. In Line 6, encodeBound, parametric in c, takes as input the current model m and returns a formula that is satisfiable if there exists a model whose cost according to c is less than that of m. This formula is conjoined with χ. Line 7 checks satisfiability of new χ. If χ is not satisfiable, then m represents a target instruction sequence with the minimum cost. In Line 8, getCode returns the instruction sequence directly represented by the model m, as described in Section 2.2.

2.2 Our Encoding

Given code p and target architecture specification a, our approach generates three kinds of constraints:

- semantics of source code p
- semantics of an arbitrary instruction sequence in the target architecture a
- observational equivalence of p and an arbitrary target instruction sequence: if the input states of p and the target sequence are equivalent, then the corresponding output states are equivalent.

A solution to the conjunction of all these constraints directly represents a target instruction sequence that correctly implements the source code.

In Line 2 of Algorithm 1, encodeCorrectness(a,p) returns the formula

$$\chi \stackrel{\text{def}}{=} \forall x, x', y, y'.\widehat{p} \wedge \widehat{a} \wedge \widehat{o}$$

where $\widehat{p}$ is a symbolic representation of p, x and x' are input and output of p, respectively, y and y' are input and output of a target sequence, $\widehat{o}$ is equivalence constraint over the observable portion of the program state, and $\widehat{a}$ is the symbolic representation of the semantics of target instruction sequences of arbitrary length:

$$\widehat{a} \stackrel{\text{def}}{=} \forall j < n. \bigwedge_{i \in I} (instr(j) = i \rightarrow \tau_i(state(y, j), state(y, j+1)))$$

where n is a free variable denoting the length of the output instruction sequence, I is the set of all possible target instructions in a and τ_i is the semantics of instruction i. Therefore, the (large) formula $\widehat{a}$ is the same for all input programs p.

An element in I represents a concrete instruction. Each opcode, condition, flags, and combination of operands defines a separate concrete instruction. For example, `ADD R0, R1, R2` and `ADD R3, R4, R5` are both in I. Effectively, I is just the set of unique identifiers of instructions in a.

For every instruction $i \in I$, the formula τ_i is satisfied by a pair of states σ, σ' if and only if the execution of i starting in σ can terminate in state σ', i.e., τ_i specifies the operational semantics of instruction i. In the presence of non-determinism (for example, allocation), τ_i may introduce an existential quantifier.

The target instruction sequence is represented using the function $instr$. For every $j = 1, \ldots, n$, $instr(j)$ denotes the instruction identifier i at program location j. Therefore, there is a direct translation from a model to an instruction sequence. The semantics of the sequence is constrained using $state$ function. For every $j = 1, \ldots, n$, $state(y, j)$ denotes the state before the instruction at program location j executes, and $state(y, j+1)$ denotes the state after the instruction at program location j executes, unless the instruction transfers control elsewhere.

Finally, equivalence constraints are of the form

$$\widehat{o} \stackrel{\text{def}}{=} (\text{rep}(x) =_o state(y, 0)) \rightarrow (\text{rep}(x') =_o state(y, n))$$

where rep is a function that maps values of the input code to those in the target code, and $=_o$ is an observational equivalence relation on states, that may take into account only portion of the state (e.g., return value and memory). In the presence of non-determinism (for example, allocation), $=_o$ introduces an existential quantifier.

Cost Functions For a standard cost function, encodeBound in Line 6 of Algorithm 1 produces the formula $n < m(n)$, where $m(n)$ is the value assigned to the logical variable n in model m.

Models A model m of χ can be directly translated to an implementation s of p for the target architecture a. The set of models of χ is exactly the set of all correct implementations of p for the target architecture a.

A model of χ represents code, not program states or aspect of the code, such as the names of register to be used with a particular instruction template in [19, 42]. This is conceptually different from existing superoptimization and synthesis approaches, as discussed in Section 2.3.

2.3 Comparison to Existing Methods

Consider pseudocode in Algorithm 2, which shows a basic superoptimizer. In every iteration of the outer loop in Lines 4-11, n increases. The inner loop in Lines 5-10 iterates over I^n, i.e., all instruction sequences of length n. For each candidate instruction sequence s, check returns PASS if and only if s passes all tests. In Line 7, encode(p,s) return formula φ that is satisfied if s does not correctly implement p. Line 8 checks satisfiability of φ. In Line 9, getModel returns a model of φ. This model is a counterexample to the assertion that s correctly implements p, i.e., cex represents an execution of s not allowed by p. In Line 10, getTests creates new tests based on cex. The algorithm uses them to avoid generating similar ones in the following iterations.

Algorithm 2 Basic superoptimization (existing)

Parameters: target architecture specification a
Input: code p
Output: code s that implements p with a shortest sequence from I

```
1:  function superoptimizer(p, I)
2:      Tests ← ∅
3:      n ← 0
4:      while true do
5:          for all s ∈ I^n do
6:              if check(s, Tests) = PASS then
7:                  φ ← encode(p, s)
8:                  if not satisfiable(φ) then return s
9:                  cex ← getModel(φ)
10:                 Tests ← Tests ∪ getTests(p, cex)
11:         n ← n + 1
```

Tests are typically employed by superoptmization and synthesis tools to quickly and cheaply eliminate some candidate instruction sequences, before calling the solver to check correctness. This is the main way in which existing methods reuse reasoning made by previous calls to the solver. Modern superoptimizers also prune the search space I^n before iteration n to avoid some obviously redundant sequences.

There are conceptual differences between encode(p,s) in Algorithm 2 and encodeCorrectness(p,a) in Algorithm 1.

First, φ encodes counterexamples while χ encodes correct code. Our approach starts the search with a correct but possibly suboptimal sequence and repeatedly calls the solver to find a better one. In contrast, existing methods start with a candidate sequence that is expect to be optimal and call the solver to check correctness. Therefore, our method can be

```
1  int sign (int x) {
2    if (x < 0) return -1;
3    if (x > 0) return 1;
4    return 0;
5  }
```

Figure 1. Running example: C code.

stopped at any time with a correct (but possibly suboptimal) solution, whereas existing methods cannot return a correct solution until the end.

Second, the size of φ depends on n, whereas the size of χ depends on the size of a. That is, the size of φ necessary increases on every iteration of the outer loop. While a is very large, it remains the same throughout Algorithm 1, whereas s significantly changes on every call to the solver in Algorithm 2. Each call to the solver in Algorithm 1 may take longer than solver calls in Algorithm 2, because χ is a more complex quantified formula, compared to φ, and the solver is expected to search a larger space. On the other hand, the solver has more opportunities to reuse reasoning in Algorithm 1, which should reduce the total time spent in the solver.

Note that as presented in Line 6 of Algorithm 1, the size of χ increases in every iteration. This is an optimization that allows the solver to reuse reasoning from previous calls. If the size of χ is too big, all the constraints produced by EncodeBound in previous iterations can be discarded, without changing the meaning of the formula, to guarantee that the size of χ in all calls to the solver is the same. This should not be necessary, because constraints produced by EncodeBound tend to be small.

Counterexample Guided Inductive Synthesis, e.g., [19, 42], use templates instead of concrete instructions to reduce the encoding size. These methods call the solver to find template parameters. We also use templates to reduce the size of our encoding of the instruction semantics. Their constraints depend on the length of the sequence of instruction templates, ours does not.

3 Constraints

In this section, we provide more details about the constraints generated by our prototype. Our encoding of LLVM IR extends [28] with the ability to handle phi nodes, inspired by symbolic execution and a symbolic encoding of out-of-ssa transformation. Our encoding of ARM ISA semantics is closely related to the two-vocabulary formulas for ISA semantics in [42], and extends it with branches.

We demonstrate these aspects of our constraints on a traditional example: the `sign` function from [29], shown in Fig. 1. Consider the corresponding LLVM IR in Fig. 2.

```
1  def i32 sign (i32 x):
2  ; <label>:L0
3    v1 = icmp slt i32 x, 0
4    br i1 v1, label L1, label L2
5  ; <label>:L1
6    br label L5
7  ; <label>:L2
8    v2 = icmp sgt i32 x, 0
9    br i1 v2, label L3, label L4
10 ; <label>:L3
11   br label L5
12 ; <label>:L4
13   br label L5
14 ; <label>:L5
15   v3 = phi ([L1, -1], [L3, 1], [L4, 0])
16   ret i32 v3
```

Figure 2. Running example: LLVM IR produced by Clang.

```
1  CMP   R0, #0
2  MOVGT R0, #1
3  MOVLT R0, #-1 ; Return value in R0
```

Figure 3. Running example: ARM Instruction Sequence.

$\widehat{L0} \leftrightarrow$	$\rho_1 = (\rho_x < 0)$ $\wedge \left((\rho_1 = \text{true}) \wedge \widehat{L1}\right) \vee \left((\rho_1 = \text{false}) \wedge \widehat{L2}\right)$
$\widehat{L2} \leftrightarrow$	$\rho_2 = (\rho_x > 0)$ $\wedge \left((\rho_2 = \text{true}) \wedge \widehat{L3}\right) \vee \left((\rho_2 = \text{false}) \wedge \widehat{L4}\right)$
$\widehat{L1} \leftrightarrow$	$\widehat{L5} \wedge (\rho_3 = -1)$
$\widehat{L3} \leftrightarrow$	$\widehat{L5} \wedge (\rho_3 = 1)$
$\widehat{L4} \leftrightarrow$	$\widehat{L5} \wedge (\rho_3 = 0)$
$\widehat{L5} \leftrightarrow$	true

Figure 4. Running Example: LLVM IR Constraints

3.1 LLVM IR Constraints

The input is a loop-free LLVM IR code fragment p, which is in SSA form [10], and the output is the constraint $\widehat{p}$. Intuitively, $\widehat{p}$ is a symbolic execution of p.

Code p consists of basic blocks, which form a (loop-free) control flow graph (CFG) with a designated entry label *lentry*. Each basic block starts with a unique label, followed by a (possibly empty) sequence of phi instructions, then a sequence of operation instructions, and ends with a terminator instruction, such as a branch or a function return.

There are 6 basic blocks in Fig. 2, labelled L0-L5, with L0 being the entry label *lentry*. Basic block at L5 has three predecessor blocks, labelled L1, L3, and L4.

Let l be a label in LLVM IR p. We use $\widehat{l}$ to denote the encoding of the basic block that starts at the label l. The encoding $\widehat{l}$ is a conjunction of the encodings of all the instructions in l. Then, $\widehat{p}$ is simply $\widehat{lentry}$.

$$\forall j < n.$$
$$instr(j) = \texttt{ADD} \rightarrow state(y, j+1)[r_d(j)] = state(y, j))[r_n(j)] + state(y, j)[r_n(j)] \wedge PRES$$
$$instr(j) = \texttt{MOV} \rightarrow state(y, j+1))[r_d(j)] = state(y, j)[r_n(j)] \wedge PRES$$
$$instr(j) = \texttt{MOVGT} \rightarrow ite(GT, state(y, j+1)[r_d(j)] = state(y, j)[r_n(j)] \wedge PRES, state(y, j+1) = state(y, j))$$

Figure 5. Running example: subset of ISA constraints in $\hat{a}$

Fig. 4 shows the constraints for each label in Fig. 2. We first explain the encoding of operation instructions, which is based on [28, 50], and then demonstrate our encoding of branch and phi instructions.

Operands Let *op* be an operand in an LLVM IR instruction, ty be its type, and n be the number of bits ty occupies. We use $\rho(op, ty)$ to denote the logical bit-vector variable of width n that represents *op* in SMT constraints.

In the example, x and $v3$ are of type $i32$, and $v1$ and $v2$ are of type $i1$. To simplify the presentation, we use the following shortcuts: $\rho_x = \rho(x, i32)$, $\rho_1 = \rho(v1, i1)$, $\rho_2 = \rho(v2, i1)$, $\rho_3 = \rho(v3, i32)$, where ρ_x and ρ_3 are bit-vectors of length 32, and ρ_1 and ρ_2 are bit-vectors of length 1.

Comparison Operations The single-bit integer type $i1$ is used in LLVM IR to represent boolean values, such as results of comparison operations *icmp* in our example (Line 3, Line 8). Comparison operations on bit-vectors in SMT return a value of sort Bool, which is distinct from the sort for bit-vectors of width 1 in SMT. Our encoding of LLVM IR comparison operations to SMT takes care of the conversion, using *if-then-else* construct *ite*. SMT bit-vector constants *#b1* and *#b0* of width 1 are used represent LLVM IR boolean constants *true* and *false*, respectively.

For example, the constraint for the operation in Line 3 is $\rho_1 = ite(bvslt(\rho_x, nat2bv(32, 0)), \#b1, \#b0)$. To simplify the presentation, we use $<$, 0, true, and false instead of *bvslt*, $nat2bv(32, 0)$, *#b1*, and *#b0*, respectively, when confusion is unlikely. The above constraint for Line 3 can be written as $\rho_1 = ite(\rho_x < 0, \text{true}, \text{false})$. We present it as $\rho_1 = (\rho_x < 0)$ for simplicity. Similarly, for Line 8, we get $\rho_2 = (\rho_x > 0)$.

Branch Instructions The conditional branch instruction at the end of block L0 (Line 4) is encoded as

$$\left((\rho_1 = \text{true}) \wedge \widehat{L1}\right) \vee \left((\rho_1 = \text{false}) \wedge \widehat{L2}\right)$$

The constraints $\rho_1 = \text{true}$ and $\rho_1 = \text{false}$ come from the case split on the value of the operand $v1$ of the conditional branch instruction. Therefore, $\widehat{L0}$ is the conjunction of the encoding of the *icmp* instruction from Line 3 and the conditional branch instruction from Line 4.

Phi Instructions For each label l_i of a phi instruction in block l, we create a constraint that expresses the corresponding assignment. If there is more than one phi instruction in l, we conjoin all assignments for label l_i. The resulting constraint is denoted $\langle l_i, l\rangle$ and used to encode $\widehat{l_i}$.

For example, from L1 alternative of the phi instruction at the beginning of L5 block in Line 15, we get the constraint $\rho_3 = -1$. This constraint, denoted $\langle L1, L5\rangle$, is used in the encoding of the unconditional branch instruction in L1:

$$\widehat{L1} \leftrightarrow \widehat{L5} \wedge (\rho_3 = -1)$$

Similarly, for L3 and L4 we get:

$$\widehat{L3} \leftrightarrow \widehat{L5} \wedge \langle L3, L5\rangle \qquad \text{where } \langle L3, L5\rangle \text{ is } \rho_3 = 1$$
$$\widehat{L4} \leftrightarrow \widehat{L5} \wedge \langle L4, L5\rangle \qquad \text{where } \langle L4, L5\rangle \text{ is } \rho_3 = 0$$

3.2 ISA Constraints

To illustrate ISA constraints, Fig. 5 shows part of $\widehat{a}$ covering ARM instructions relevant to our example: ADD, MOV, MOVGT. The conditions are a combination of values of designated CPSR register bits. For example, $GT \stackrel{\text{def}}{=} Z = 0 \vee N = V$ where Z is $state(y, j)[CPSR][30]$, N is $state(y, j)[CPSR][31]$, and V is $state(y, j)[CPSR][28]$.

Templates We use templates to reduce the size of the ISA encoding, similarly to the way templates are used in [19, 42]. A template t represents a subset of instructions from I using uninterpreted functions:

- opcode: $instr(j)$
- immediate operand: $c_n(j)$ maps program location j to an n-bit constant
- register operands: $r_d(j)$, $r_n(j)$, $r_m(j)$, and $r_o(j)$ map program location j to register name for the destination d and operands n, m, o
- branch destination $brdest(j)$ denotes the target program location for a branch instruction at program location j

For example, template $instr(j) = \texttt{ADD}$, $r_d(j) = 0$, $r_n(j) = 1$, $c_8(j) = 5$ represents the instruction `ADD R0, R1, #5`.

The semantics of MOV updates the destination register and preserves all other components of the state. Preserve constraints for templates involves an additional quantifier:

$$PRES \stackrel{\text{def}}{=} \forall r.r \neq r_d(j) \rightarrow state(y, j+1))[r] = state(y, j)[r]$$

3.3 Equivalence Constraints

In our example, there is one input x and the return value is v_3, stored in register R0. We get the following formula for $\widehat{o}$.

$$\text{rep}(\rho_x) = state(y, 0)[R0] \rightarrow \text{rep}(\rho_3) = state(y, n)[R0]$$

3.4 From Model to Code

When SMT solver determines that its input formula is satisfiable, it returns a model that satisfies it. The model contains

an assignment to all free variables and uninterpreted functions. In our encoding, SMT model will assign n, $instr$, r_d, r_n, c_4, c_8, $brdest$ when relevant. SMT Solver API provides a way to inspect the model to get value of the above variables. To convert SMT Model to ARM assembly, we inspect the model m: for each $j = 0$ to n get template instruction identifier from $instr(j)$ in m, then get the appropriate operands for template instruction from $r_d(j)$, $r_n(j)$, $r_m(j)$, $r_o(j)$, $c_8(j)$.

4 Preliminary Results

We report on experiments we performed to check whether Z3 can solve complex constraints that arise from our approach.

We used Hacker's Delight [47] benchmarks, which have been used in [19, 35] to evaluate their synthesis and superoptimization method. We used Clang to generate LLVM IR from the original C code of these benchmarks. Our prototype takes LLVM IR as input and generates ARMv7-A assembly as output. We used the standard cost function, i.e., the length of the generated instruction sequence. Our prototype successfully generated optimal code for 17 of the 25 benchmarks in that suite, in less than 30 minutes per benchmark, and took up to 4 iterations of Algorithm 1 per benchmark. The remaining 8 benchmarks require additional improvements of the prototype.

We evaluated the performance of code generated by our prototype in comparison to `gcc -O3` on a Raspberry Pi 3 system containing a 4 Core Broadcom BCM2837 ARMv8 Cortex A53 1.2 GHz with 1 GB RAM. Each of the above benchmarks are executed for 4 million iterations. Fig. 6 presents the speedups achieved by our prototype over `gcc`.

We discuss the benchmarks for which our prototype generates code that is substantially different from the code generated by `gcc`. Fig. 7 shows the LLVM IR for `p01`, `p13`, and `p16`. For these three benchmarks, both `gcc` and Stoke [35] generate a naive translation of the input program.

For `p01`, our prototype generates a more efficient code with an AND instruction, because using shifter operands in a data processing instruction AND is faster than subtraction.

For `p13`, which works similarly to Massalin's example [29] to find the sign of the input argument, our prototype generates conditional move instructions.

For `p16`, which finds maximum of two inputs, `gcc` generates 4 instructions, whereas our prototype generates only 2 instructions: compare and conditional move.

5 Related Work

Production compilers are hand-tuned to generate efficient code for target hardware, but do not guarantee optimality even for straight-line code. Compilation time must be low for the compiler to be usable in a standard way. Programmers can control compilation time vs quality of generated code using few predefined optimization levels (e.g., `-O0` through `-O3` in gcc), and some additional combinations of compiler options. Our approach is currently slower in terms of compilation time. However, our approach can generate increasingly better code as the time allotted for compilation increases, with a finer control over this trade-off.

Massalin's superoptimization [29] exhaustively enumerates sequences of instructions of increasing length, testing each for equivalence with the input code. There is no guarantee of correctness, but this approach is highly unlikely to generate wrong code. The cost function is therefore the length of the generated sequence. The approach can handle only straight-line code as input and output, and did not handle operations that involve memory.

Denali [21] can handle only straight-line code as input and output, including memory operations, but supports elaborate cost functions that take into account parameters of modern microarchitectures (such as multiple issue, memory latency, etc). At a high level, our encoding is similar to theirs in that they also generate a first order formula that expresses the meaning of the source code and the semantics of the entire ISA, including cost function. The difference is in the approach taken to search for a model of this formula. Denali uses a specialized solver based on equality-preserving transformations designed by an expert. Unlike Denali, our approach is guide the search from within an SMT solver, integrating with existing and future SMT solver technology. This approach also makes it easier to reason about soundness, in comparison to the proof of soundness of hand-written transformation in Denali.

Stoke [35–38] is a modern superoptimizer based on stochastic search. Markov Chain Monte Carlo sampler rapidly explores the search space of all possible target programs to find one that is an optimization of the input program. Stoke handles control flow, including loops, using data-driven equivalence checking algorithm [37] even in the presence of input constraints [38]. Their algorithm conjectures a simulation relation based on the observed data, and then uses this relation to formulate an equivalence constraint that can be discharged by an SMT solver. Our approach uses SMT solver to (implicitly) derive an appropriate simulation relation for loop-free fragments. We do not handle loops. Stoke was extended in [36] to Floating-Point. We have not experimented with floating point or complex data-types yet.

GreenThumb [32] applies different search techniques in parallel. One of the techniques introduced in [32] uses enumerative search to rapidly prune away invalid candidate instruction sequences and selectively refine the abstraction under which candidates are considered equivalent via an incremental use of test cases. Counterexamples have previously been used for inductive synthesis of loop-free programs, e.g., [19, 20, 40, 42]. Our approach shifts the search for optimal code into the SMT Solvers. It would be interesting to explore the use of counter examples and abstraction refinement within the solver for handling quantifiers that arise from our approach.

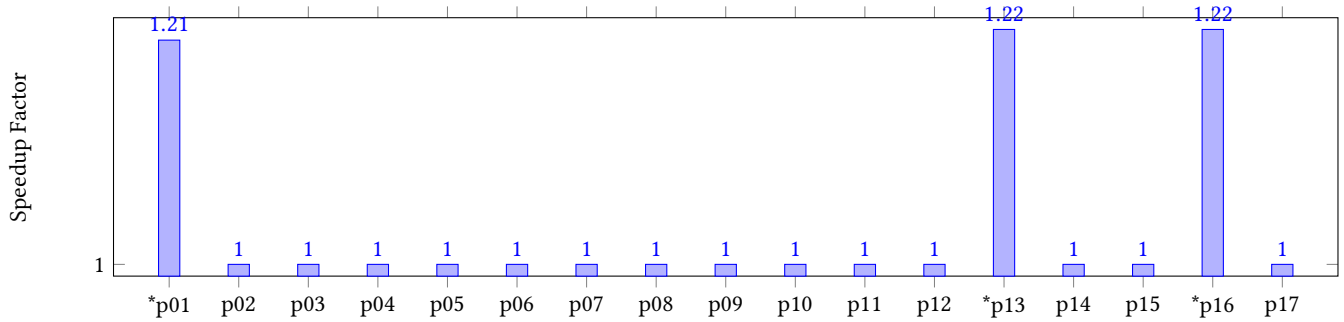

Figure 6. Speedups for benchmarks compiled by our technique over gcc. Benchmarks for which our technique discovered an algorithmically distinct rewrite are annotated with a star.

	p01	p13	p16

Input LLVM IR	`def i32 p01 (i32 x):`
	`  o1 = sub i32 x, 1`
	`  res = and i32 x, o1`
	`  return res`
		`def i32 p13 (i32 x):`
		`  o1 = ashr i32 x, 31`
		`  o2 = sub i32 0, x`
		`  o3 = lshr i32 o2, 31`
		`  res = or i32 o1, o3`
		`  return res`
			`def i32 p16 (i32 x, i32 y):`
			`  o1 = xor i32 x, y`
			`  o2 = uge i32 x, y`
			`  o3 = sub i32 0, o2`
			`  o4 = and i32 o1, o2`
			`  res = xor i32 o3, y`
			`  return res`

Output ARM assembly	`AND R0, R0, R0, LSL #1`	`CMP  R0, #0`
		`MOVGT R0, #1`
		`MOVLT R0, #-1`
			`CMP  R1, R0`
			`MOVGT R0, R1`

Figure 7. ARM code produced by our prototype for p01, p13, and p16 benchmark

Superoptimization boils down to proving a form of program equivalence. In our setting, this is slightly non-standard because the target program is unknown and its size is unbounded. Proving (standard) program equivalence even without loops is not decidable in the presence of unbounded memory. Therefore, there is source code for which a solver will not find a solution, or will not improve on a solution even when a better solution exists.

Differential program equivalence methods [22, 23] that use SMT Solvers and can handle loops have been successfully applied in the setting of software security. [45] can also handle certain loops. Instead of encoding Compiler's intermediate representation as a formula and using an existing solver, [45] constructs an intermediate representation of all target instructions sequences that correctly implement the source. Then, it analyzes the intermediate representation to find an optimal sequence. The construction works by applying specifically designed equality preserving transformations. It has been applied to generate Java Bytecode. The proof of correctness is non-trivial.

To avoid explicit loop unrolling during model checking, [30] uses SMT theory of Lists to express unbound program executions. This is similar in spirit to our encoding of ISA. Instead of lists, we use a function *instr* that takes into account instruction size and alignment to handle some interesting costs, e.g., does the loop body fit into the loop buffer, is the target of a branch word aligned. Another concern is whether a solver can handle a combination of lists and bit vectors (and arrays for expressing memory).

Binary translation tools such as [9, 24, 46] typically decode a binary to an intermediate representation, optimize it, and generate the target binary. Binary translation is used for example to improve the performance of existing emulation tools, emulate multiple source-target architecture pairs, and provide security guarantees. A static binary translator in [41] is based on peephole superoptimizations. Our approach can also be used in this context.

Recently, [39] demonstrated the effectiveness of constraint-based software analysis, in particular due to the use of MaxSAT. Our approach, viewed as constraint-based software synthesis, can similarly benefit from the emerging technology of MaxSMT.

6 Conclusion and Future Work

We aim to change the way CPU-specific optimizations are implemented in production compilers. The new compiler architecture will generate code that takes full advantage of the capabilities of new microarchitectures, without modifying the compiler. It will provide greater flexibility for hardware design and faster deployment of new hardware, as well as better software performance and system reliability.

We describe the core of the new approach to code generation and optimization, and present some encouraging results. Unlike existing superoptimization and synthesis methods, our approach shifts the entire search problem into the solver. This provides a way to reuse reasoning and guide the solver using domain specific information about the input program and the target architecture. We believe that tighter integration with the solver is the right direction for this problem domain.

Our encoding takes advantage of unbounded models of first-order logic. However, quantifiers are used in a way that takes the constraints outside of a decidable fragment. The first challenge is to keep the constraints within first-order logic, without fixing a-priori the length of the target instruction sequence. The second challenge is to reason about such constraints automatically and efficiently. We described a way to address the first challenge, and demonstrated that satisfiability checking of these quantified formulas is feasible in simple cases. The next steps are to evaluate the scalability in comparison to existing tools and the effectiveness of existing SMT solvers.

A preliminary prototype of our approach takes LLVM IR as input and employs Z3 SMT solver to generate ARMv7-A assembly. The prototype supports a small subset of LLVM IR and ARMv7-A ISA. This prototype enables us to experiment with a variety of cost functions and encodings. To speed up code generation, we are evaluating modern SMT features such as tactics, quantifiers, and constraint optimization. We are also integrating the prototype with a (more) complete ISA specification, including floating point and vector instructions. We plan to support other architectures (ARMv8-A, x86_64, and PowerPC) using existing formal ISA specifications.

Acknowledgements

We thank Eva Darulova and the anonymous reviewers for their feedback on drafts of this paper.

References

[1] ARM. 2017. ISA specification for ARMv8-A. (2017). https://developer.arm.com/products/architecture/a-profile/exploration-tools, released in April 2017.

[2] Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 394–403.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Int. Conf. on Computer Aided Verification (CAV)*. 171–177. http://cvc4.cs.nyu.edu/web/.

[4] Clark W. Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore D. Zuck. 2005. TVOC: A Translation Validator for Optimizing Compilers. In *Int. Conf. on Computer Aided Verification (CAV)*. 291–295.

[5] Nikolaj Bjørner and Mikolás Janota. 2015. Playing with Quantified Satisfaction (short paper). In *Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. 15–27.

[6] Nikolaj Bjørner and Nina Narodytska. 2015. Maximum Satisfiability Using Cores and Correction Sets. In *Int. Joint Conf. on Artificial Intelligence (IJCAI)*. 246–252.

[7] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. *νZ* - An Optimizing SMT Solver. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 194–199.

[8] Martin Brain, Tom Crick, Marina De Vos, and John P. Fitch. 2006. TOAST: Applying Answer Set Programming to Superoptimisation. In *Int. Conf. on Logic Programming (ICLP)*. 270–284.

[9] Derek L. Bruening. 2004. *Efficient, Transparent and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. K Zadeck. 1988. *An Efficient Method of Computing Static Single Assignment Form*. Technical Report. Providence, RI, USA.

[11] Bjorner N. de Maura L. 2008. Z3: An efficient SMT solver. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[12] Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Conference on Automated Deduction (CADE)*. 183–198.

[13] Leonardo Mendonça de Moura and Grant Olney Passmore. 2013. The Strategy Challenge in SMT Solving. In *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*. 15–44.

[14] Shaked Flur, Kathryn E Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *ACM Symp. on Principles of Programming Languages (POPL)*. 608–621.

[15] Anthony Fox. 2012. Directions in ISA Specification. Interactive Theorem Proving.

[16] Andreas Fröhlich, Armin Biere, Christoph M Wintersteiger, and Youssef Hamadi. 2015. Stochastic Local Search for Satisfiability Modulo Theories. In *Proc. of Association for the Advancement of Artificial Intelligence (AAAI)*. 1136–1143.

[17] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin V. Bonilla, John Thomson, Christopher K. I. Williams, and Michael F. P. O'Boyle. 2011. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming* 39, 3 (2011), 296–327.

[18] Shilpi Goel, Warren A. Hunt Jr., Matt Kaufmann, and Soumava Ghosh. 2014. Simulation and formal verification of x86 machine-code programs that make system calls. In *Formal Methods in Computer-Aided Design (FMCAD)*. 91–98.

[19] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*. 62–73.

[20] Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. 2010. A Simple Inductive Synthesis Methodology and Its Applications. In *Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 36–46.

[21] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-directed Superoptimizer. In *SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*. 304–314.

[22] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Int. Conf. on Computer Aided Verification (CAV)*. 712–717.

[23] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential assertion checking. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. 345–355.

[24] Monica S. Lam (Ed.). 2000. *SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*. ACM.

[25] Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *ACM Symp. on Principles of Programming Languages (POPL)*. ACM Press, 42–54. http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf

[26] Junghee Lim, Akash Lal, and Thomas W. Reps. 2011. Symbolic analysis via semantic reinterpretation. *Int. Journal on Software Tools for Technology Transfer (STTT)* 13, 1 (2011), 61–87.

[27] Junghee Lim and Thomas W. Reps. 2013. TSL: A System for Generating Abstract Interpreters and its Application to Machine-Code Analysis. *Trans. on Prog. Lang. and Syst. (TOPLAS)* 35, 1, Article 4 (2013).

[28] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*. http://web.ist.utl.pt/nuno.lopes/pubs.php?id=alive-pldi15.

[29] Henry Massalin. 1987. Superoptimizer: A look at the smallest program. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[30] Aleksandar Milicevic and Hillel Kugler. 2011. Model Checking Using SMT and Theory of Lists. In *Nasa Formal Methods Symposium*, Vol. 6617. Springer Verlag, 282–297.

[31] George C. Necula. 2000. Translation validation for an optimizing compiler. In *PLDI*. 83–94.

[32] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling Up Superoptimization. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 297–310.

[33] Alastair Reid. 2016. Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture. In *Formal Methods in Computer-Aided Design (FMCAD)*.

[34] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Int. Conf. on Computer Aided Verification (CAV)*. 198–216.

[35] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 305–316.

[36] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic Optimization of Floating-point Programs with Tunable Precision. In *SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*. 53–64.

[37] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven Equivalence Checking. In *Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 391–406.

[38] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2015. Conditionally Correct Superoptimization. In *Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 147–162.

[39] Xujie Si, Xin Zhang, Radu Grigore, and Mayur Naik. 2017. Maximum Satisfiability in Software Analysis: Applications and Techniques. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. 68–94.

[40] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 404–415.

[41] Alex Aiken Sorav Bansal. 2008. Binary Translation Using Peephole Superoptimizers. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[42] Venkatesh Srinivasan and Thomas Reps. 2015. Synthesis of machine code from semantics. In *SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*. http://research.cs.wisc.edu/wpis/papers/pldi15.pdf.

[43] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-Based Translation Validator for LLVM. In *Int. Conf. on Computer Aided Verification (CAV)*. 737–742.

[44] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2016. A new verified compiler backend for CakeML. In *Int. Conf. on Functional Programming (ICFP)*. 60–73.

[45] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: a New Approach to Optimization. In *ACM Symp. on Principles of Programming Languages (POPL)*. 264–276.

[46] David Ung and Cristina Cifuentes. 2000. Machine-adaptable Dynamic Binary Translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO '00)*. 41–51.

[47] H. S. Warren. 2002. *Hacker's Delight*. Addison-Wesley.

[48] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. 2014. Approximations for Model Construction. In *Int. Joint Conf. on Automated Reasoning (IJCAR)*. 344–359.

[49] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. 2016. Deciding Bit-Vector Formulas with mcSAT. In *Theory and Applications of Satisfiability Testing (SAT)*. 249–266.

[50] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *ACM Symp. on Principles of Programming Languages (POPL)*. 427–440.