Finding a few needles in some large haystacks: Identifying missing target optimizations using a superoptimizer

Hal Finkel
Argonne National Laboratory
hfinkel@anl.gov

Abstract

A superoptimizer is a program that searches for an optimal sequence, often the shortest sequence, of instructions that implement some set of operations. Early superoptimizers used exhaustive searches, relying on testing a large number of trial inputs to assess equivalence. Modern superoptimizers, like the one described here, often use Satisfiability Modulo Theories (SMT) solvers to prove equivalence for all inputs.

1. What is a superoptimizer?

A superoptimizer is a program that searches for an optimal sequence, often the shortest sequence, of instructions that implement some set of operations. Early superoptimizers used exhaustive searches, relying on testing a large number of trial inputs to assess equivalence. Modern superoptimizers, like the one described here, often use Satisfiability Modulo Theories (SMT) solvers to prove equivalence for all inputs.

2. What is an SMT solver?

Informally, an SMT solver is a program that attempts to prove or disprove, a mathematical formula stated using terms and predicates, that there exists an assignment of values to the variables that makes the formula true.

3. CVC4

CVC4 is a BSD-licensed, extensible, SMT solver:

- Many built-in theories (rationals and integer linear arithmetic, arrays, tuples, records, inductive data types, bit-vectors, and equality over uninterpreted functions)
- A command-line interface and also a C++ API
- Available from: http://cvc4.cs.nyu.edu/web/

A simple example:

Please provide me with a specific counter-example.

```cvc4`

- COUNTERMODEL:
  - b = 1;
  - i: INT = 0;
```  

What if I assert that x is always positive, then what?

```cvc4`

- QUERY x > 0;
```  

4. A real example

Let's validate r168594, an addition to ValueTracking's isKnownToValue function, which says, if x and y are known to be non-zero powers of two, then:

```
(add x, (x, y))
```

is also a non-zero power of two:

```cvc4`

- OPTION produc modele:
  - x: BITVECTOR(32);
  - y: BITVECTOR(32);
  - z: BITVECTOR(32);
  - assert BVPLUS(32, x, y) = z;
  - assert x = 0x1000000000000000;
  - assert y = 0x1000000000000000;
  - assert z = 0x2000000000000000;
  - assert x = 0x1000000000000000;
```  

Now we have a value for i that holds for the provided random inputs. Verify it for all inputs:

```cvc4`

- QUERY b = 0x0000000000000003;
```  

5. Solving for satisfying constants

For building a superoptimizer, we often want to be able to ask whether there exist some fixed values of a set of constants that make a formula generally true. How can this be done? Let's find b such that:

```
(add x, (x, y)) = b + x
```

```cvc4`

- OPTION produc modele:
  - x: BITVECTOR(32);
  - y: BITVECTOR(32);
  - assert BVPLUS(32, x, y) = b;
  - assert x = 0x1000000000000000;
  - assert y = 0x1000000000000000;
```  

6. Modeling 64-bit PowerPC in CVC4

Creating CVC4 functions that correspond to the PPC64 fixed-point instructions is fairly straightforward:

```cvc4`

- LAMBDA (ra, rb : BITVECTOR(64)): BVMULT(64, SX(ra[31:0],64), rb);
- LAMBDA (ra : BITVECTOR(64), si : BITVECTOR(16)): BVMULT(64, ra, SX(si, 64));
- LAMBDA (si : BITVECTOR(16)): SX(si, 64);
- LAMBDA (ra, rb : BITVECTOR(64), si : BITVECTOR(16)): BVMULT(64, ra, SX(si, 64));
- LAMBDA (si : BITVECTOR(16)): SX(si, 64);
- LAMBDA (ra, rb : BITVECTOR(64), si : BITVECTOR(16)): BVMULT(64, ra, SX(si, 64));
- LAMBDA (ra : BITVECTOR(64), si : BITVECTOR(16)): BVMULT(64, ra, SX(si, 64));
- LAMBDA (ra, rb : BITVECTOR(64), si : BITVECTOR(16)): BVMULT(64, ra, SX(si, 64));
- LAMBDA (si : BITVECTOR(16)): SX(si, 64);
```  

7. Building the superoptimizer

The superoptimizer reads from assembly files, tracking register dependencies, looking for trees of single-user instructions. Why? Because if a tree of single-user instructions has a simpler replacement, then that is almost always preferable and implementable as an optimization somewhere in the compiler. Then:

- For each single-user tree, translate the tree into a CVC4 expression
- Generate all possible (shorter) alternatives with the same inputs and the same output type
- Combine these alternatives into a large parametrized "switch statement"
- Use CVC4 to search for a set of input constants, and a value of the parameter that selects the alternative, that allows proving equivalence between the original tree and the alternative for all input values.

8. What does it find?

Sometimes we find simple missing patterns:

```
     (x | y) -> (x | y)
```

Sometimes we find more complicated things:

```
     (x & y) -> (x & y)
```

9. What then?

From most likely to least likely:

- Improve instruction selection, peephole optimization, spill-code generation, etc.
- Implement target-specific DAG combines
- Improve IR-level optimizers