Finding a few needles in some large haystacks:

Identifying missing target optimizations using a superoptimizer

Hal Finkel

Argonne National Laboratory

hfinkel@anl.gov

Abstract

So you're developing an LLVM backend, and you've added a bunch of TableGen patterns, custom DAG combines and other lowering code; are you done? This poster describes the development of a specialized superoptimizer, applied to the output of the compiler on large codebases, to look for missing optimizations in the PowerPC backend. This superoptimizer extracts potentially-interesting instruction sequences from assembly code and then uses the open-source CVC4 SMT solver to search for provably-correct shorter alternatives.

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 relations from some set of well known *background theories*: real numbers, integers, bit vectors, arrays, lists, etc.

3. CVC4

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

- Many built-in theories (rational 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:

```
CVC4> OPTION "incremental";
CVC4> OPTION "produce−models";
```

If I have two integers, x and y, are they always equal?

```
CVC4> x, y : INT;
CVC4> QUERY x = y;
invalid
```

Please provide me with a specific counter-example.

```
CVC4> COUNTERMODEL;
x : INT = −1;
y : INT = 0;
```

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

```
CVC4> ASSERT x >= 0;
CVC4> QUERY x = y;
invalid
CVC4> COUNTERMODEL;
x : INT = 0;
y : INT = 1;
```

4. A real example

Let's validate r185954, an addition to ValueTracking's isKnownToBeAPowerOfTwo function, which says, if x and y are known to be non-zero powers of two, then

$$(add\ nsw\ x,\ (and\ x,\ y))$$

is also a non-zero power of two:

```
CVC4> OPTION "produce−models";
CVC4> x, y : BITVECTOR(32);
CVC4> ISPOW2 : BITVECTOR(32) −> BOOLEAN = LAMBDA(x :
  BITVECTOR(32)): BVPLUS(32, x, 0hexffffffff) & x = 0hex00000000 AND x
  /= 0hex00000000;
CVC4> ASSERT ISPOW2(x);
CVC4> % assert nuw or nsw
CVC4> ASSERT (BVZEROEXTEND(BVPLUS(32, x, x & y), 1) = BVPLUS
  (33, x, x & y)) OR (SX(BVPLUS(32, x, x & y), 33) = BVPLUS(33, x, x & y));
CVC4> QUERY(ISPOW2(BVPLUS(32, x, x & y)));
valid
```

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 $f + f + f = b * f$:

```
CVC4> OPTION "produce−models";
CVC4> b, f : BITVECTOR(64);
```

First, generate a bunch of random inputs:

```
CVC4> fa : BITVECTOR(64) = 0hex0b46a8f39e73154b;
CVC4> fb : BITVECTOR(64) = 0hex0a490d5cf77a2c00;
CVC4> fc : BITVECTOR(64) = 0hex644fd6d5edd990f2;
...
```

then assert that the formula holds for them:

```
CVC4> ASSERT BVPLUS(64, BVPLUS(64, fa, fa), fa) = BVMULT(64, fa, b);
CVC4> ASSERT BVPLUS(64, BVPLUS(64, fb, fb), fb) = BVMULT(64, fb, b);
CVC4> ASSERT BVPLUS(64, BVPLUS(64, fc, fc), fc) = BVMULT(64, fc, b);
...
CVC4> CHECKSAT;
sat
```

In this special "satisfied" context, we can extract details of the satisfying solution by asking for a counter-example of the "false" query:

```
CVC4> QUERY FALSE;
invalid
CVC4> COUNTEREXAMPLE;
...
b : BITVECTOR(64) = 0hex0000000000000003
...
```

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

```
CVC4> ASSERT b = 0hex0000000000000003;
CVC4> QUERY BVPLUS(64, BVPLUS(64, f, f), f) = BVMULT(64, f, b);
valid
```

6. Modeling 64-bit PowerPC in CVC4

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

```
addi: (BITVECTOR(64), BITVECTOR(16)) −> BITVECTOR(64) =
        LAMBDA (ra : BITVECTOR(64), si : BITVECTOR(16)):
          BVPLUS(64, ra, SX(si, 64));
li: BITVECTOR(16) −> BITVECTOR(64) =
        LAMBDA (si : BITVECTOR(16)): SX(si, 64);
...
mulli: (BITVECTOR(64), BITVECTOR(16)) −> BITVECTOR(64) =
         LAMBDA (ra : BITVECTOR(64), si : BITVECTOR(16)): BVMULT(64,
           ra, SX(si, 64));
mullw: (BITVECTOR(64), BITVECTOR(64)) −> BITVECTOR(64) =
         LAMBDA (ra, rb : BITVECTOR(64)): BVMULT(64, SX(ra[31:0],64),
           SX(rb[31:0],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:

```
xor(r4,li(−1)) −> nand(r4,r4)
...
mulld(r18,li(88)) −> mulli(r18,i_0_0)
  where:
    i_0_0 = 88
...
```

Sometimes we find more complicated things:

```
cmpw(extsb(r7),extsb(r7)) −> cmpld(r7,r7)
...
rldicr(clrldi(r6,32),2,61) −> rldic(r6,i_0_0,i_0_1)
  where:
    i_0_0 = 2
    i_0_1 = 30
...
isel(r6,r5,cmplwi(or(rlwinm(r7,29,31,31),rlwinm(r4,30,31,31)),0),2) −> r5
...
cmpldi(isel(r4,r3,cmplwi(or(rlwinm(r6,29,31,31),rlwinm(r5,30,31,31)),0),2),0)
  −> cmpldi(r3,i_0_0)
  where:
    i_0_0 = 0
...
rldicr(clrldi(rlwinm(r4,29,31,31),32),2,61) −> rlwinm(r4,i_0_0,i_0_1,i_0_2)
  where:
    i_0_0 = 31
    i_0_1 = 29
    i_0_2 = 29
...
```

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

2013 LLVM Developers' Meeting, 6-7 November 2013, San Francisco, CA