Superoptimizers
A Look at the Smallest Program

By

Henry Massalin

Presented By

Jay Patel

Problem Statement

- Find an optimal algorithm for compiling a function.
  - What does optimal mean?
    - Speed
    - Correctness
    - In this case, assembly output should be small as possible

Superoptimizer

- Given as input a set of instructions and a compiled function. The superoptimizer finds the smallest program that is equivalent to the input function.

- Most critical part of the tool is to determine if two pieces of code are equivalent
  a) Probabilistic Test
  b) Pruning

Cool Example 1

- A function that returns the sign bit of X.

```
if(x>0) return 1;
else if( x < 0) return -1;
else return 0;
```

- Typical assembly code

```
      le d0,0
      branch L1
      return 1
L1: ge d0,0
      branch L2
      return -1
L2: return 0
```

- Superoptimizer

```
add d0,d0
subx.1 d1,d1
negx.1 d0
addx.1 d1,d1
```

Cool Example 2

- A function that returns the minimum of (x,y)

```
if(x<y) return x;
else
    return y;
```

- Typical assembly code

```
    ge d0,d1
    branch L1
    return d0
L1: return d1
```

- Superoptimizer

```
Sub.1 d1,d0 (x-y)
subx.1 d2,d2
and.1 d2,d0
add.1 d1,d0
```

- This can be faster on a pipelined architecture where there might be a delay for a jump

Cool Example 3

- A function that returns the absolute value of x

```
if(x>=0) return x;
else
    return -x;
```

- Typical assembly code

```
    lt d0,0
    branch L1
    return d0
L1: neg d0
    return d0
```

- Superoptimizer

```
Move.1 d0,d1
add.1 d1,d1
subx.1 d1,d1
eor.1 d1,d0
sub.1 d1,d0
```

High Level Algorithm

- Choose a subset of the instruction set and store in a table for reference
- Superoptimizer consulates this table and generates all combinations of instructions
- Start with a instruction sets of length 1, length 2, etc.
- Test whether each program is equivalent to the original program
- Halt when an equivalent program is found.

Equivalence Test - Probabilistic Test

- Run the generated program on a select few smart inputs.
- If the output matches the source program, run the generated program through the entire test suite.
- Example inputs for signum
  a) Negative number, positive number, zero
- These test cases rules out programs that
  a) Return same value regardless of input
  b) Answers of the same sign
  c) Return their argument

Pruning Search Space

- No need to evaluate sequences that cannot occur in a optimal solution
- Any longer sequences that have the same effect as shorter sequences cannot be part of the optimal
  a) move x,y; move x,y

  b) and x,y; move z,y

- N-dimensional bit tables
  - For i<N, test a sequence of length i by accessing the ith dimension of the bit tables.
  - For each instruction in the sequence, if the lookup value is 1, no need to continue along this search path.

Limitations

- Even with optimizations, still an exponential search
  - 12 instruction functions take hours to find optimal solution
- Pointer verification is hard. Since pointers can point anywhere, all possibilities need to be checked (means all possible memory addresses)
  - On a 256-byte machine 2^(2^(256*8)) possible pointer locations
  - Probabilistic test is inconclusive with pointers
- Machine dependence, the input instruction set must be able to run on the machine Superoptimizer is running on.

Applications

- Superoptimizer is useful for

1) Helping the compiler find optimizations for little tasks like
   a) multiplication by a constant. (E.g. array indexing)
   b) Checking for equality between two values.
2) Equivalent sets of programs generated by Superoptimizer can aide the compiler in peephole optimization
3) Assembly programmers can use Superoptimizer to reduce the size of critical stdlib functions
   a) Author gives example of rewriting printf function in 500 bytes.

Conclusion

- Superoptimizer is a tool that can be used to find optimal(shortest) equivalent programs.
- Even though it uses an exhaustive search
  a) Probabilistic test and Pruning help reduce the time it takes to find the optimal solution.
  b) Once equivalent optimal programs are found for common tasks they can be stored.
- The most interesting result is the relationship between arithmetic and logical instructions. And the ability they have to simulate branching instructions.
- Also, search structure lends itself to parallelism, slave processes can check for equivalence for a particular program and the input program. And report back to the master process.