Paper Summary
Superoptimizer:
A Look at the Smallest Program

by Jay Patel

1. **Overview**

   The paper purposes a tool called Superoptimizer that tries to find the shortest length assembly language compiled version of a function (maybe also be refered to as a program). The tool tries to create a compiled version of the function that is shorter in length than the compiler would produce. It takes advantage of convoluted bit shifting features of some instructions to create smaller compiled functions. Typical these functions are not readable by programmers and require some understanding to see what is exactly happening.

   The input to Superoptimizer is an already compiled function and a subset of instructions supported by the machine architecture (could be all instructions also). The Superoptimizer will try to generate the smallest possible compiled function that is equivalent to the input function based on the instruction set. Basically, Superoptimizer's algorithm is to test all possible combinations in the instruction set and find the smallest one that is equivalent to the input. It does this by generating sets of instructions of lenght 1, length 2, etc. until a optimal solution is found.

   There are two key routines in Superoptimizer. The first is quickly testing if the input function and the generated function are equivalent. Superoptimizer utilizes the Probabilistic Test to quickly determine if two functions are equivalent. The other key routine is generating a combination of instructions from the subset to create a possible equivalent function. Clearly, there are some combinations of instructions that can be quickly ruled out. Thus, Superoptimizer prunes the search space of the instruction set by ruling out combinations that cannot be part of the optimal solution.

2. **Detailed Example**

   Since compilers are used in general purpose compilation. And furthermore written by humans, it can be difficult to utilize all features of an instruction set to create optimal compiled functions. Take the following example, a function that returns the sign of a number X.

   ```
   int function(int x) {
     if(x>0) return 1;
     else if( x < 0) return -1;
     else return 0;
   }
   ```

   Typical assembly code for this routine would look as follows.

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

   Below is the optimized code that Superoptimizer was able to generate. The paper contains details on what is happening on each instruction. But below, along with the instruction, the state of all relevant registers and flag bits are shown for each instruction. Assume x is initially in the register d0.

Instruction	d0 > 0	d0 = 0	d0 < 0	Comment
add d0,d0	c=0	c=0	c=1	set the carry flag based d0's value
subx d1,d1	d1=0	d1=0	d1=-1	d1 - (d1 - carry bit)
negx d0	c=1	c=0	c=1	negx only sets the carry bit if d0 was nonzero
addx d1,d1	d1=1	d1=0	d1=-1	d1 + d1 + carry bit

3. **Probabilistic Test - Checking for program equivalence**

   In their first attempt, the authors tested for equivalence by representing the output as a boolean expression based on the input. However, arithmetic operations have on the order 2^{31}, conceivably because all possible values must be checked. Clearly, a boolean expression with this many terms is computationally and space intensive. In their initial version, Superoptimizer could only handle functions with a maximum of 3 instructions. And it could test 40 programs per second for equivalence.

   In contrast, this version of Superoptimizer uses a probablistic test. In contrast, this version of Superoptimizer can handle functions with 12 instructions and can test 50,000 programs per second for equivalence. However, strickly speaking the probabilistic test can lead to incorrect solutions (while the boolean verification cannot). Though in the authors experience the probability of finding a false positive is very low. Furthermore, the author still has to see a program that is incorrect based on the probabilistic test.

   The probabilistic test at its core is very simple. The programmer selects a couple well crafted inputs for the program being tested. For the example of the functin that returns the sign of a integer (above), the input test cases could be a negative number, a positive number and 0. After Superoptimizer generates a possible equivalent program, Superoptimizer first checks whether the two functions being compared have the same output for the select few inputs. If the possible program passes these test cases, then the rest of the test cases are tested. This reduces the number of programs that need to be tested against the entire input test suite.

   It is important to note that this methoding of testing for equivalence is not as conclusive as the boolean test; however, the author notes he has not seen an incorrect test produced with the probabilistic test.

4. **Pruning - reducing the search space**

   This part of the paper was unclear on some details that could have proven useful. As with the probabilistic test, the idea behind pruning is quite straightforward. Since Superoptimizer is testing all possible combinations, it is important to try and prune the search space as much as possible. What the author purposes is that Superoptimizer should not test any programs for equivalence that include sequences that are known not to be part of the optimal solution.

   For example, the optimal equivalent program could not contain the pair of instructions AND x,y;MOV x,y. Clearly, the result of the AND that is stored in Y, would be subsequently overwritten by moving the value of x into y. Therefore, we could replace the two instructions with the single move instruction. Thus, any time it is clear that the path being searched has a non-optimal subsequence, there is no need to check this path. Since another path in recursive search will be searched without this path.

   The author purposes using an N-dimensional bit vector structure for this task when searching for possible equivalent programs. N being the length of the longest sequence(identified by the programmer) that is known to be suboptimal. The author does not give a lot of detail past this about how the structure would work. But I believe the author is purposing to keep a bit vector for each sequence that is known to be suboptimal. Recall, during Superoptimizer execution, all sets of size 1 are tested, then all sets of size 2, etc. Thus, whenever we create a new set from i-1 to i, the sequence with i instructions should be tested if it now contains a suboptimal sequence. This is done by each instruction in the i^{th} sequence testing against one of the dimensions of the N-dimensional bit vector. As the author states, a lookup of one means the sequence is supoptimal and there is no need to go searching down this path anymore. Presumably a lookup of 1 means that all instructions tested against the one dimension tested to be part of a suboptimal sequence.

   It should also be noted, the author makes no claims about the registers involvedduring a suboptimal sequence comparison. Using the example given above, while the sequence AND x,y;MOV z,y is clearly suboptimal. The sequence AND x,b;MOV z,y still could be legal. However, in the authors description, there is no mention of checking instruction operands during a suboptimal check. Though, clearly in this case the operands do matter.

5. **Applications and Limitations**

   Finally, the author goes over the applications and limitations. Clearly in terms of limitations, Superoptimizer is still doing an exponential search which causes it only to run currently on functions with 12 instructions or less. Also, pointers present a problem because like addition and subtraction, they have a large set of possible values. And the probabilistic test on them is inconclusive. Also, the instruction subset that the Superoptimizer works with must be able to run on the native machine. Thus, for every new architecture to run Superoptimizer a code change is required.

   There are a couple benefits to Superoptimizer. First, being helping a compiler with peephole optimizations. Once, Superoptimizer runs and finds optimized solutions to common tasks a compiler deals with. For example, finding a constant offset for an array lookup. Or doing a comparison between two values. The compiler can use the set of optimal generated programs by Superoptimizer to search for sequences in an actual program and patch them in with the optimal solution. A second benefit is for assembly programmer writers that wish to implement stdlib functions as optimally as possible to reduce the space they take up. The author gives the example of optimizing the printf routine in the c stdlib. Finally, analyzing optimal programs can help architecture designers understand what instructions are useful and which ones can be discarded when designing possible instruction sets.

   Lastly, though not mentioned in the paper, I feel that the Superoptimizer program structure lends itself to parallelism. Each program that is tested for equivalence can be given off to a slave processor to do the computation. Once it finds that the program it tested was suboptimal, the master process can give it another program to test. This could help test a larger set of instruction sets and functions that are even longer.

6. **Conclusion**

   The ideas of probabilistic test and pruning described in this paper are useful tactics that can probably be used in other areas of computer science. Also, if combined with a parallel architecture they could even be faster. To the general reader of this paper, this is probably the most important thing to take away. Though, for compiler designers, systems programmers and architecture designers, the subtle interplay between logical and arithmetic operators in existing instruction sets is very compelling and could lead to other useful results.