Superoptimizer -- A Look at the Smallest Program

Henry Massalin

Department of Computer Science
Columbia University
New York, NY 10027

Abstract

Given an instruction set, the superoptimizer finds the shortest program to compute a function. Startling programs have been generated, many of them engaging in convoluted bit-fiddling bearing little resemblance to the source programs which defined the functions. The key idea in the superoptimizer is a probabilistic test that makes exhaustive searches practical for programs of useful size. The search space is defined by the processor's instruction set, which may include the whole set, but it is typically restricted to a subset. By constraining the instructions and observing the effect on the output program, one can gain insight into the design of instruction sets. In addition, superoptimized programs may be used by peephole optimizers to improve the quality of generated code, or by assembly language programmers to improve manually written code.

1. Introduction

The search for the optimal algorithm to compute a function is one of the fundamental problems in computer science. In contrast to theoretical studies of optimal algorithms, practical applications motivated the design, implementation, and use of the superoptimizer. Instead of proving upper or lower bounds for abstract algorithms, the superoptimizer finds the shortest program in the program space defined by the instruction set of commercial machines, such the Motorola 68000 or Intel 8086.

The functions to be optimized are specified with programs written using the target machine's instruction set. Therefore, the input to the superoptimizer is a machine language program. The output is another program, which may be shorter. Since both programs run on the same processor, with a well-defined environment, we can establish their equivalence.

A probabilistic test and a method for pruning the search tree makes the superoptimizer a practical tool for programs of limited size (about 13 machine instructions).

In section 2, we describe an interesting example to illustrate the superoptimizer approach. The design and algorithms used in the superoptimizer are detailed in section 3. We discuss the applications and limitations of the superoptimizer in section 4. In section 5, we compare the superoptimizer with related work. The conclusion in section 6 is followed by a list of interesting minimal programs in appendix I.

2. An Interesting Example

We begin with an example to show what superoptimized code looks like. The instruction set used here, as in most of the paper, is Motorola's 68020 instruction set. Our example is the *signum* function, defined by the following program:

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

This function compiles to 9 instructions occupying 18 bytes of memory on the SUN-3 C compiler. Most programmers when asked to write this function in assembly language would use comparison instructions and conditional jumps to decide in what range the argument lies. Typically, this takes 8 68020 instructions, although clever programmers can do it in 6.

It turns out that by exploiting various properties of two's complement arithmetic one can write *signum* in four instructions! This is what superoptimizer found when fed the compiled machine code for the signum function as input:

```
(x in d0)
add.l   d0,d0    |add d0 to itself
subx.l  d1,d1    |subtract (d1 + Carry) from d1
negx.l  d0       |put (0 - d0 - Carry) into d0
addx.l  d1,d1    |add (d1 + Carry) to d1
(signum(x) in d1) (4 instructions)
```

Like a typical superoptimized program, the logic is really convoluted. One of the first things that comes to mind is "where are the conditional jumps?". As we will see later, many functions that would normally be written with conditional jumps are optimized into short programs without them. This can result in significant speedups for certain pipelined machines that execute conditional jumps slowly.

Let us see how it works. The "add.l d0, d0" instruction doubles the contents of register d0, but more importantly, the sign bit is now in the carry flag. The "subx.l d1, d1" instruction computes "d1-d1-carry --> d1". Regardless of the initial value of d1, d1-d1-carry is -carry. Thus d1 is -1 if d0 was negative and 0 otherwise. Besides negating, "negx.l d0" will set the carry flag if and only if d0 was nonzero. Finally, "addx.l d1, d1" doubles d1 and adds the carry. Now if d0 was negative, d1 is -1 and carry is set, so d1+d1+carry is -1, if d0 was 0, d1 is 0 and carry is clear, so d0+d0+carry is 0, if d0 was positive, d1 is 0 and carry is set, so d1+d1+carry is 1.

3. Superoptimizer Internals

Superoptimizer takes a program written in machine language as the input source. It finds the shortest program that computes the same function as the source program by doing an exhaustive search over all possible programs. The search space is defined by choosing a subset of the machine's instruction set, and the op-codes of these instructions are stored in a table. Superoptimizer consults this table and generates all combinations of these instructions, first of length 1, then of length 2, and so on. Each of these generated programs is tested, and if found to match the function of the source program, superoptimizer prints the program and halts.

Two methods are used to reduce the search time. The first is a fast probabilistic test for determining the the equivalence of two programs. The second is a method for pruning the search space while maintaining the guarantee of optimality. These two methods will now be discussed, but first a boolean-logic equivalence test will be explained, which was the first test proceedure implemented, because it finds use in the tree pruning method.

3.1. Boolean Test

The most important part of superoptimizer is the routine that determines whether two pieces of code computes the same function. The first version of superoptimizer used what we call the *boolean program verifier*. The idea was to express the function output in terms of boolean-logic operations on the input argument. Once this is done, two programs are equivalent if their boolean expressions matches minterm for minterm.

In practice, some instructions such as *add* and *mul* have boolean expressions with on the order of 2^31 minterms. Various methods had been devised to reduce the memory requirements, but it took too long to compute the boolean expressions for every program generated. The initial version of superoptimizer tested about 40 programs per second, and this allowed programs of up to 3 instructions to be generated in reasonable time.

3.2. Probabilistict Test

The idea behind the probabilistic test is simple: run the machine code for the program being tested a few times with some set of inputs and check whether the outputs match those of the source program. The idea here is that most programs will fail this simple test, and a full program verification test will be done only for the few programs that this test fails to catch. Running through a few carefully chosen test vectors takes very little time. Currently, superoptimizer can test 50000 programs per second and the exhaustive search approach becomes practical.

The test vectors are chosen (manually) to maximize the probability that a random program will fail on the first or second test. For example, the test vectors for the *signum* function included -1000, 0 and 456 as the first three vectors. This quickly eliminates programs that return the same answer regardless of argument, answers of the same sign, as well as programs that return their argument. Following these vectors, all the numbers from -1024 to 1024 were tested.

It was found in practice that a program has a very low probability of passing this execution test and failing the boolean verification test. This fact proves very useful since most programs of interest have boolean expressions that are too large to fit in memory. We can dispense with the boolean test and manually inspect the generated programs for correctness, without having to analyze a large number of wrong programs. This manual check is not difficult since the programs are small (about 4 to 13 instructions). Currently, superoptimizer runs without the boolean check, and the author has yet to find an incorrect program.

One problem introduced by the probabilistic execution test is machine dependency. The test works only if the instruction set being searched can be executed on the machine running the superoptimizer. In other words, if we wish to change the instruction set, we would have to port the superoptimizer to the new machine. This port is not too difficult since the current version of superoptimizer is rather short (about 300 lines of 68020 assembly code), however it does require that one translate it into the target assembly code.

3.3. Pruning

In order to further reduce the search time, we filter out instruction sequences that are known not to occur in any optimal program. Any sequence of instructions that has the same effect on the machine state as a shorter sequence cannot be part of an optimal program, because if it were, you can get a shorter program by substituting the shorter sequence, and therefore the program was not optimal. Typical sequences include the obviously silly "move X,Y; move X,Y" and "move X,Y; move Y,X", "and X,Y; move Z,Y" in which the MOVE destroys the result of the AND, "and #0,X" which does the same thing as "clr X", and "and X,Y; <any> Z,W; and.l X,Y" where the second AND is superfluous.

This filtering is done with N-dimensional bit tables, where N is the length of the longest sequence we wish to filter. Each instruction in the sequence we wish to test indexes one dimension of the bit table, and a lookup value of '1' causes the program to be rejected as non-optimal (and also as incorrect, since it is the same as a shorter program, and superoptimizer has already checked all shorter programs).

There are two ways that these bit tables can be filled. A human can tell the bit table maker program to exclude all "move X,Y; move Y,X" sequences. The program then scans all instructions in all dimensions of the bit matrix and sets the values accordingly. One can also run superoptimizer with the boolean test, and have it find the equivalences on its own.

4. Applications and Limitations

4.1. Current Limitations

Even with the probabilistic test, the exhaustive search still grows exponentially with the number of instructions in the generated program. The current version of superoptimizer has generated programs 12 instructions long in several hours running time on a 16MHz 68020 computer. Therefore, the superoptimizer has limited usefulness as a code generator for a compiler.

Another difficulty concerns pointers. A pointer can point anywhere in memory and so to model a pointer in terms of boolean expressions one needs to take all of memory into account. Even on a 256-byte machine, there are 2^(2^(256*8)) possible minterms, and these are just too many. We have explored the probabilistic test approach for pointers, but the results have been inconclusive.

Currently, we have only the 68020 version of the superoptimizer running the probabilistic test, so the instruction sets are restricted to subsets of the 68020 set. The machine-independent version of superoptimizer is limited to very short programs.

4.2. Applications

Because of the pointer problem, superoptimizer works best when the instruction set is constrained to register-register operations. Even so, it can be used to analyze instruction sets. Some of the programs in appendix I were tried on the Western Electric WE32000 microprocessor and in every case the resulting program was longer than the 68020 programs. The reason for this was found to be the lack of an add-with-carry instruction and the fact that the flags are set according to the 32 bit result, even for byte sized operands. The National Semiconductor NS32032 was also found to suffer from flag problems. Here the difficulty is that extra instructions are needed to test the outcome of an operation because few instructions set the flags.

Another use would be in the design of RISC architectures. One can try various instruction sets simply by coding their function in terms of boolean expressions and seeing what superoptimizer comes up with. A particular instruction may be omitted if superoptimizer finds a short equivalent sequence of other instructions.

The superoptimizer may be very useful in optimizing little tasks that often confront a compiler. An example is finding the optimal program that multiplies by a particular constant for use in accessing arrays and such. Some examples of multiplication by constants can be found in I.6.

Another useful feature of superoptimizer is the identity tables containing the equivalent program sequences found. These programs may be extracted and used to increase the power of a conventional peephole optimizer.

In practice, the best use of superoptimizer has been as an aid to the assembly language programmer. An experienced programmer can use superoptimizer to come up with nifty equivalent sequences for small sections of his code, while retaining the overall logical flow that makes a program maintainable. This method has been used by the author (along with another program that optimizes code emulating state machines) to write the C library function *printf* in only 500 bytes.

5. Comparison with Related Work

The most commonly used optimization techniques are those that attempt to improve the code that a compiler produces. Examples are peephole optimizers and data-flow analysis. Peephole optimizers [2] are table driven pattern matchers that operate on the assembly language code produced by the compiler. Every time a sequence of instructions is matched by one of the tables, a smaller and faster replacement sequence is used.

Data-flow analysis [1] is a technique applied during the semantic and code generation phases of the compilation process. It improves code in several ways. First, it eliminates redundant computations (common sub-expression elimination). Second, it moves expressions within a loop whose values do not depend on the loop variable to outside the loop (loop invariance). Third, (also in a loop) it converts expressions of the form 'K * loop-index' into the equivalent arithmetic progression 'TMP = TMP + K' (strength reduction).

These methods are general. They work regardless of the machine-specific details such as the representation of an integer. However, usually the result is not optimal in either space or speed. Superoptimizer depends on the instruction set, however, the code is guaranteed to be optimal in space and it does a very good job in speed as well.

Krumme and Ackley [4] have written a code generator for the DEC-10 computer that is based on exhaustive search. Their method translates each interior node of an expression tree into several viable instruction sequences. These sequences are then pieced together to form a set of translations for the entire expression. This set is then searched to find the cheapest alternative.

In their method, there is a one to one correspondence between the instructions in the translation and the original expression. For example, if there's an add in the expression, there will also be an add somewhere in the generated code. Superoptimizer has a more global view of the problem. It 'translates' one sequence of instructions into another completely different sequence. On the other hand, superoptimizer can't translate large programs.

The two approaches can be seen as complementing each other. Superoptimizer can be used to prepare the code generation tables used in Krumme and Ackley's method. Their method can also be incorporated into superoptimizer to increase the size of programs that can be handled. Superoptimizer can generate several short equivalent sequences for small fragments of the source program, and then Krumme and Ackley's method would be used to piece these together and find a short overall sequence.

Kessler [3] has written a code optimization tool, which translates sequences of instructions into one single instruction. The superoptimizer can be seen as a more general tool with broader applications, since it can transform programs of many instructions to another one of several instructions. However, Kessler's optimizer works regardless of program size, and therefore can be easily used to optimize compiled code. Another difference is that he uses template matching, while superoptimizer relies on exhaustive search.

6. Conclusion

We have taken a practical approach to the search for the optimal program. We have found that the shortest programs are surprising, often containing sequences of instructions that one would not expect to see side by side. The *signum* function is an example of this, and the *min* and *max* functions given in section I.3 contain a beautiful combination of the logical *and* and the arithmetic *add*.

Exhaustive search is justified by these results, and a probabilistic test allows programs of practical size to be produced. Although results are limited to a dozen instructions, those found are already useful. Many examples of these can be found in Appendix I.

One of the most interesting results is not the programs themselves, but a better understanding of the interrelations between arithmetic and logical instructions. Similar ideas seem to come up consistently in the superoptimized programs. These include the sequence '*add.l d1,d1; subx.l d1,d1*' that extracts the sign of a number in the *signum* and *abs* functions and the sequence '*sub.l d1,d0; and.l d2,d0; add.l d1,d0*' that selects one of two values depending on a third in the *min* and *max* functions.

In the future, we hope to explore these ideas further, and compile a list of useful arithmetic-logical idioms that can be concatenated to form optimal or near-optimal programs.

Appendix

I. More Interesting Results

I.1. *SIGNUM* Function

The *signum* function has been defined in section 2. Given the 68000 instruction set, four is the minimum number of instructions to compute *signum*. Interestingly, three suffice on the 8086.

```
(x in ax)
cwd     (sign extends register ax into dx)
neg  ax
adc  dx,dx
(signum(x) in dx)
```

I.2. Absolute Value Function

Find the absolute value of a number, excluding conditional jumps from the instruction set.

```
(x in d0)
move.l   d0,d1
add.l    d1,d1
subx.l   d1,d1
eor.l    d1,d0
sub.l    d1,d0
(abs(x) in d0)
```

Notice that although it is longer than the classical method (test; jump-if-positive; negate), it has no jumps! This might actually be faster than the classical method on some pipelined machines where jumps are expensive.

I.3. Max and Min

This program finds the maximum of the unsigned numbers in d0 and d1 and returns the answer in d0. The comments on the right show what's in the various registers during execution and is similar to the boolean expression checker's method of analysis.

```
(d0=X, d1=Y)  |Flag,Reg|If d1>d0   |If d1<=d0
sub.l    d1,d0|(C,d0) =|(1, X-Y)   |(0, X-Y)
subx.l   d2,d2|(C,d2) =|(1,11..11)|(0,0...0)
or.l     d2,d0|(C,d0) =|(1,11..11)|(0,X-Y)
addx.l   d1,d0|d0 =    |Y          |X
(d0 = max(X, Y))
```

This program finds the minimum of the unsigned numbers in d0 and d1 and returns the answer in d0.

```
(d0=X, d1=Y)  |Flag,Reg|If d1>d0   |If d1<=d0
sub.l    d1,d0|(C,d0) =|(1, X-Y)   |(0, X-Y)
subx.l   d2,d2|d2 =    |111...111  |000...000
and.l    d2,d0|d0 =    |X-Y        |0
add.l    d1,d0|d0 =    |X          |Y
(d0 = min(X, Y))
```

Simultaneous min and max.

```
(d0=X, d1=Y)  |Flag,Reg|If d1>d0   |If d1<=d0
sub.l    d1,d0|(C,d0) =|(1, X-Y)   |(0, X-Y)
subx.l   d2,d2|d2 =    |111...111  |000...000
and.l    d0,d2|d2 =    |X-Y        |0
eor.l    d2,d0|d0 =    |0          |X-Y
add.l    d1,d0|d0 =    |Y          |X
add.l    d2,d1|d1 =    |X          |Y
(d0 = max(X, Y), d1 = min(X, Y))
```

I.4. Logical Tests

Here are some logical tests that yield true/false answers. Sequences such as these have immediate application in a compiler to improve execution speed. Shown here are the tests for zero and non-zero.

```
Suitable for BASIC     Suitable for C, PASCAL

d0 =  0 if d0 == 0     d0 = 0 if d0 == 0
   = -1 if d0 != 0        = 1 if d0 != 0
neg.l    d0            neg.l       d0
subx.l   d0,d0         subx.l      d0,d0
                       neg.l       d0


d0 = -1 if d0 == 0     d0 =  1 if d0 == 0
   =  0 if d0 != 0        =  0 if d0 != 0
neg.l    d0            neg.l    d0
subx.l   d0,d0         subx.l   d0,d0
not.l    d0            addq.l   1,d0
```

By prepending 'move.l A,d0; sub.l B,d0' to the above one can construct tests for A == B and A != B.

I.5. Decimal to Binary

This piece converts a 8 digit BCD number stored in d0, one digit to a nibble, to binary with the result also in d0. It is the longest sequence ever generated by superoptimizer, and was actually done in three stages. The idea that this was even possible came while generating sequences to multiply by 10. At first I had superoptimizer compute the 2 digit BCD to binary conversion function '*((d0 & 0xF0) >> 4) * 10 + (d0 & 0x0F)*'. This came out surprisingly short:

```
(2 digit BCD number in d0)
move.b   d0,d1
and.b    #$F0,d1
lsr.b    #3,d1
sub.b    d1,d0
sub.b    d1,d0
sub.b    d1,d0
(binary equivalent in d0)
```

What is actually being computed is

```
          ans = d0 - 3 * ((d0 & 0xF0)/8)
```

Representing the contents of d0 as (H:L) where H is the upper nibble and L is the lower nibble we get

```
          d0 = 16 * H + L,     d0 & 0xF0 = 16*H
          ans = (16*H+L) - 3 * (16*H/8)
              = 16*H+L - 6*H
              = 10*H + L
```

which is the 2 digit BCD to binary function. Encouraged by this result, superoptimizer was put to the task of computing first the 4 digit BCD to binary function and then the 8 digit BCD to binary function. Here is the 8 digit converter:

```
(8 digit BCD number in d0)
move.l   d0,d1                       *
and.l    #$F0F0F0F0, d1              *
lsr.l    #3,d1                       *
sub.l    d1,d0                       *
sub.l    d1,d0                       *
sub.l    d1,d0                       *
move.l   d0,d1                       +
and.l    #$FF00FF00,d1               +
lsr.l    #1,d1                       +
sub.l    d1,d0                       +
lsr.l    #2,d1                       +
sub.l    d1,d0                       +
lsr.l    #3,d1                       +
add.l    d1,d0                       +
move.l   d0,d1                       -
swap     d1                          -
mulu     #$D8f0,d1                   -
sub.l    d1,d0                       -
(binary equivalent in d0)
```

What is most amazing is the first section (marked by * alongside the program) It looks exactly like the 2 digit BCD to binary function. This section computes 4 simultaneous 2 digit BCD to binary functions on adjacent pairs of nibbles and deposits the answer back into the byte occupied by those nibbles. The second part (marked by +) computes two simultaneous 2-byte base 100 to binary conversion functions. Finally, the third part computes the function 'high-word-of-d0 * 10000 + low-word-of-d0' to complete the conversion.

I.6. Multiplication by Constants

During a two week period, superoptimizer was used to find minimal programs that multiply by constants. A sampling of these programs is included in this section.

An interesting observation is that the average program size increases as the multiplication constant increases, but it increases very slowly. The average size of programs that multiply by small numbers (less than 40) is 5 instructions, most programs that multiply by numbers in the hundreds are 6 to 7 instructions long, and programs that multiply by thousands are between 7 and 8 instructions long.

```
d0 *= 29                         d0 *= 39
move.l   d0,d1                   move.l   d0,d1
lsl.l    #4,d0                   lsl.l    #2,d0
sub.l    d1,d0                   add.l    d1,d0
add.l    d0,d0                   lsl.l    #3,d0
sub.l    d1,d0                   sub.l    d1,d0
```

```
d0 *= 156
move.l  d0,d1
lsl.l   #2,d1
add.l   d1,d0
lsl.l   #5,d0
sub.l   d1,d0
```

```
d0 *= 625
move.l  d0,d1
lsl.l   #2,d0
add.l   d1,d0
lsl.l   #3,d0
sub.l   d1,d0
lsl.l   #4,d0
add.l   d1,d0
```

I.7. Division by Constants

Division turns out to be difficult to optimize. A general divide by constant that works for all 32-bit arguments is too long to realize any time gain over the divide instruction, and is certainly not shorter. Additionally, there doesn't seem to be any nifty arithmetic-logical operations that simplify the process. The generated programs just multiply by the reciprocal of the constant. Since we do an exhaustive search, this negative result can be seen as a confirmation of the inherent high cost of divisions for the instruction sets considered.

The following programs were generated in an attempt to gain insight into binary to BCD algorithms, another area where superoptimizer has had little success. Note that even with the restricted argument range, these are much longer than the multiply programs.

```
d0 = trunc(d0/10)       for d0 = 0..99
move.b d0,d1
add.b  d0,d0 |d0 = 10 * x
lsr.b  #1,d1 |d1 = .1 * x
add.b  d1,d0 |d0 = 10.1 * x
lsr.b  #3,d0 |d0 = .0101 * x
add.b  d1,d0 |d0 = .1101 * x
lsr.b  #3,d0 |d0 = .0001101 * x

d0 = trunc(d0/100)      for d0 = 0..9999
move.w d0,d1
lsr.w  #1,d1 |d1 =   .1 * x
add.w  d0,d0 |d0 = 10 * x
add.w  d0,d1 |d1 = 10.1 * x
lsr.w  #5,d0 |d0 =   .0001 * x
add.w  d1,d0 |d0 = 10.1001 * x
lsr.w  #8,d1 |note: you can't lsr.w #10,d1
lsr.w  #2,d1 |d1 =   .00000000101 * x
sub.w  d1,d0 |d0 = 10.10001111011
lsr.w  #8,d0 |d0 = .0000001010001111011 * x
```

References

[1] Aho, A.V., Sethi, R, Ullman, J.D.
Compilers Principles, Techniques, and Tools.
Addison Wesley, 1986.

[2] Davidson, J.W. and Fraser, C.W.
Automatic Generation of Peephole Optimizations.
In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 111-116.
ACM/SIGPLAN, June, 1984.

[3] Kessler, P.B.
Discovering Machine-Specific Code Improvements.
In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 249-254.
ACM/SIGPLAN, June, 1986.

[4] Krumme, D.W. and Ackley, D.H.
A Practical Method for Code Generation Based On Exhaustive Search.
In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 185-196.
ACM/SIGPLAN, June, 1982.