Writing Efficient C Code for Optimizers
Oliver Bradley, SAS Institute Inc., Cary, NC

Today's C compilers are highly optimizing. When you use such a compiler, many of the traditional ways of hand-optimizing your program no longer apply. But don't expect the compiler to do it all on its own. There are many things you can do to help an optimizing compiler speed up your program. This paper discusses some of these things. Most of these techniques apply to any optimizer, and the paper discusses these first. Then I'll move on to some techniques that are specific to the SAS/C and Lattice C Compilers that we market.

Note that these techniques apply specifically to C. Some of them may apply to other languages, but every language has its own rules. The techniques also assume a highly optimizing compiler — not all languages currently have such a compiler.

TECHNIQUES THAT APPLY TO ANY OPTIMIZER

Avoid Aliasing

The optimizer does best when it has the most information. Consider the following loop:

```c
int i, a, *p, *q;
/* other code ... */
for (i = 0; i < n; i++)
  p[i] = q[i];
```

*i* is an induction variable (in optimizer terminology). While *i* needs to be multiplied by the size of an int to be used as an index, an optimizer can replace that multiplication by an addition because it knows exactly what happens to *i* on every iteration through the loop.

Now let's suppose you need to call another function, one that takes the address of an int, earlier in the program. You decide to use *i*, since it isn't used for anything else at that place in the program, and you add the following statement:

```c
f2(t.i);
```

The for loop can no longer be optimized. *i* must be accessed in memory at every iteration and multiplied by 4 for use as an index. Why is this? It's because the optimizer no longer knows what happens to *i* in the loop. Its address may have been assigned to a pointer, and *p* might point to it. It's no longer safe for the optimizer to assume that *i* is only changed by the for statement. Of course *p* probably doesn't ever point to *i*, but an optimizer needs to be sure. Hoping that something doesn't happen is a sure way to generate incorrect code. To avoid this deoptimization, you need to use a different variable in the call to the f2 function.

This simple example illustrates a very important point. An optimizer can do far more when it knows exactly what can happen to a variable. Aliasing — making it possible for a variable to be changed in indirect ways — can be a tremendous hindrance. Taking the address of a variable is an obvious way to cause it to be aliased. Some other ways to avoid aliasing problems are:

- Avoid using external variables.
- Avoid using static variables.
- Use lvalue casts to make a variable look like a pointer.
- Add lvalue casts to static variables when using a function that changes the value of such variables.
- Avoid using external variables.

An external variable (one declared with the extern keyword or defined outside of any function) can be accessed from anywhere in the program. Your program might contain functions not in the current compilation, and those functions might take the address of the external variable and assign it to a pointer. The optimizer cannot be sure (unless it does cross-module analysis); it has to assume that external variables are aliased.

- Avoid using static variables.
- Use lvalue casts to make a variable look like a pointer.
- Add lvalue casts to static variables when using a function that changes the value of such variables.
- Avoid using external variables.

These are not as big a problem as external variables but can still cause deoptimization. If you call another function, the optimizer has to assume that static variables might be changed. This is because C allows recursive calls, and therefore, the other function could call back to the present function, and it could change the variable.

- Avoid heavy pointer access to unchanged values.
- Avoid using external variables.
- Use lvalue casts to make a variable look like a pointer.
- Add lvalue casts to static variables when using a function that changes the value of such variables.
- Avoid using external variables.

A value accessed through a pointer is automatically aliased since the optimizer normally has no idea where in memory it is. Such values have to be reaccessed from memory if something might have changed them since the last access. This is particularly inefficient if several levels of indirection are involved. Thus, if your program makes heavy use of pointer access to values that do not change, copy the value or values to a local (auto) variable (and don't take its address, of course!). By doing this, you tell the optimizer that this value doesn't change.

- Limit use of (char *) pointers.
- Use lvalue casts to make a variable look like a pointer.
- Add lvalue casts to static variables when using a function that changes the value of such variables.
- Avoid using external variables.

The ANSI C Standards committee wrote the C aliasing rules to minimize the deoptimizations caused by aliasing. An optimizer is obliged to consider only type-based aliasing. That is, an int can only be changed through a pointer to int (including accessing an int structure member), a double can only be changed through a pointer to double, and so on. This works well. But the ANSI committee had to make an exception for functions such as memcpy that copy blocks of memory and may well change any type of data. The committee used the (char *), or pointer to character, type for this. A pointer to character is allowed to point to any type of data. Thus, dereferencing a character pointer, or passing one to a function, might change the value of any aliasable value since the pointer is allowed to point to it. An optimizer must allow for this possibility. And you should allow for it by restricting the use of character pointers to the places where you really need them. For example, it's much better to use an assignment statement to copy a structure, rather than a byte-by-byte loop, even though the optimizer might be able to optimize the loop.

Avoid goto

The goto statement can present a lot of problems to an optimizer. Every goto you use makes your function more complex. Some optimizers will not optimize a function that is too complex. Even one that optimizes any function, as our optimizer does, will take longer to analyze a more complex function than a simpler one.

goto and labels also inhibit good code generation. When a goto is encountered, most variables must be stored back to memory. (The most highly used variables can be kept in registers.) Similarly, when a code generator sees a label, it must forget about most variables that it holds in registers because code that branches to the label may give them a different value. This is a problem with any type of control flow, but the problem is worst for goto because it is the most unstructured construct (that is, it can go anywhere).
Use const and Avoid volatile

`const` and `volatile` are the ANSI twins - new keywords that are used to give the compiler additional information about values and memory locations. Both keywords can be used in the same places, but otherwise they are very different. `const` is very good for optimization because it tells the optimizer that the variable’s value never changes. If a `const` variable is initialized in the same compilation, the optimizer can simply replace uses of the variable with the constant value. Even for `const` variables initialized in another compilation, `const` enables the optimizer to avoid worrying about changes to the value and aliasing situations. The value can be kept in a register in more situations, leading to better codes. You can also declare pointers to `const` variables, telling the optimizer that the pointed-to value doesn’t change.

`volatile` is almost the exact opposite of `const`. `volatile` tells the compiler that the variable or memory location may change in ways it doesn’t know about. Consequently, the compiler is obliged to access a `volatile` value in memory every time it is used. It can never retain the value in a register. Nor can the optimizer make any assumptions about what value the variable has or assume that the value is unchanged since the last `C` expression that changed it. In essence, `volatile` tells the compiler that it’s not safe to do any optimizations on that variable. So you shouldn’t expect any optimizations to be done on a `volatile` value, even obvious ones. Use `volatile` only when you need the value to be accessed in memory every time.

Floating-Point Arithmetic

If your program uses floating-point arithmetic a lot, it’s worth giving some attention to optimization. There are a number of factors that make it difficult for an optimizer to optimize floating-point arithmetic, and often, recoding can help the optimizer.

The first thing to be aware of are the rules for associative rearrangement. If you coded the following, where `u` and `v` are unsigned integers:

```c
e = 2 + (v - 10000);
```

you’d expect the optimizer to fold the constants and just subtract 9998 from `v`, and most optimizers would. For the equivalent floating-point expression, `d` and `e` being doubles:

```c
d = 2.0 + (e - 10000.0);
```

the ANSI C Standard requires the optimizer to honor your parentheses. The reason is that `e` might be close to 10000.0, and subtracting 10000.0 first, then adding 2.0, may yield extra precision in the final result. The ANSI rules assume that you know what you’re doing and that you’ll put the parentheses where you need them.

Of course, this means that it’s up to you to regroup and rearrange floating-point expressions where they need to be optimized. Be especially careful of parentheses added by macros: the compiler can’t distinguish them from those you code yourself and is obliged to honor them all.

Let’s consider another example:

```c
double a, b, c;
int i, n;
// ...
for (i = 0; i < n; i++)
c[i] = a/b;
```

`a` and `b` don’t change within the loop (assuming no aliasing problems) so you might expect an optimizer to move the calculation of `a/b` outside the loop. It won’t (or perhaps I should say it shouldn’t). Why? If `b` and `n` are both zero, moving `a/b` outside the loop would cause a division by zero that would not have occurred in the original program. It’s not safe for the optimizer to move code if moving might create an exception that would not otherwise have occurred. If you know it’s safe to move `a/b` out of the loop, you need to do it yourself.

Another area worth attention is function calls. Except in some special cases, an optimizer can’t do a lot with a function call. It doesn’t know what the called function does and has to assume the worst. So the optimizer generally can’t move function calls around or optimize them away. This has serious implications for floating-point calculations. An optimizer will try hard to keep values for floating-point calculations in floating-point registers. But on a function call, all the floating-point registers have to be saved, and restored when the called function returns. This can be a very expensive operation. On some hardware, such as Intel’s 80x87 floating-point chips, loading and storing the registers takes a lot more time than arithmetic on the registers. While the optimizer and code generator may be able to avoid storing all the floating-point registers in some cases, it’s much better to avoid the function call completely. Again, this is something you need to do. As far as possible, move function calls out of regions of intense floating-point calculation, particularly loops. Arrange your program logic so that function calls can be made before or after the calculations. If you call a function to return a series of values for use in a calculation loop, consider calculating all the values at one time and storing them in an array.

Common Subexpressions

A common subexpression is an expression that is used more than once in a function. For example, you might refer to `a+b` in several different places.

Before the advent of optimizing compilers, a common technique for optimizing your program yourself was to assign common subexpressions to temporary variables. You’d code:

```c
    c = a+b;
```

and use `c` in place of `a+b`.

Using an optimizer, you may wonder if it’s still necessary, or even harmful, to do this since optimizers recognize common subexpressions and can do it for you. Unfortunately, the answer is “sometimes.” Sometimes you still need to assign common subexpressions to temporary variables, sometimes you don’t, and sometimes it’s actually harmful. Here are some general rules which will help.

- Don’t create your own temporary variable if the optimizer (or optimizers) you’re using will recognize the common subexpression. Your source code will be easier to follow, in most cases, with the original expression. And the optimizer will keep track of any situations in which it is not correct to reuse the expression, such as a change to one of the variables in the expression.
- Assume that the optimizer will recognize simple, commutative rearrangements. All optimizers of which I am aware will recognize `a+b` and `b+a` as the same subexpression, so you’d no longer use `c` as in the example above.
- Don’t assume that associative rearrangement will be recognized. For example, the SAS optimizer currently recognizes only some simple cases involving constants or address offsets. Don’t expect an optimizer to recognize `a+b` as a common subexpression in `c=a+(b+1)` and `b+(g+h+(a+1))`, for example. For floating-point expressions, remember that the ANSI Standard prohibits most associative rearrangements, as I discussed earlier.
TECHNIQUES SPECIFIC TO SAS COMPILERS

Use a Medium Function Size

The SAS Institute Global optimizer analyzes only the current compilation. That is, it does not currently do intermodule or interprocedural optimizations (except for in-lining, discussed below). This means that if you keep your functions very small, the optimizer doesn’t have much to work with. A small function is normally straightforward portable code because they may change to the C source code. This will help with any compiler, not just the SAS/C Compiler.

You may wonder why larger functions aren’t recommended. The optimizer will optimize them too, of course, but generally you would see the same improvement per line of code as for a medium-sized function. The disadvantage of large functions is that more memory and CPU time are required to optimize them. Sometimes memory and CPU usage increase in a nonlinear way with function size (this depends on how complex the function is). Thus while large functions will be optimized, you may find it expensive.

Use In-Line Functions

In-lining is an enhancement that has been added to our optimizer since last SUGI (for the 370, it’s in Release 4.50 of the SAS/C Compiler products). In-lining allows you to generate the code for small functions in-line, in place of a function call. This avoids all function call overhead. It also opens up more opportunities for the optimizer. For example, if some of the parameters in the particular call are constants, tests on those parameters in the called function become tests of constants when it is in-lined. Such tests can be done at compile time, leading to the elimination of some of the code from the called function. (The SAS/C Compiler and Library Users’ Guide contains an example). Other cross-optimizations may be possible when the in-line code is considered as part of the outer function.

Again, in-lining is something that doesn’t just happen automatically. It requires some work on your part. First, let’s assume that the function or functions you want to in-line are already in the same source file as its callers. You can

- use the INLOCAL option to in-line static functions that are called only once.
- use the COMPLEXITY option to in-line all functions up to a certain size. Here, you need to consider the trade-off between execution speed and code size. The larger the function that is in-lined, the more the code size increases.
- use the __inline option for in-line specific functions that you select, regardless of their size.
- use the Depth and DEPTH options to inhibit or control recursive in-lining.

When the function or functions you want to in-line are not in the same source file as the calls to them, you also need to make the function or functions available to the compiler to get them in-lined. Generally, the easiest way to do that is to copy the entire function to a header file and add the __inline keyword to it. Again, you need to choose which functions to do this for, and you’ll want to be sensitive to the time/space trade-off.

Don’t Spend Time Choosing Register Variables

Our current optimizer always chooses register variables for you. It ignores any register declarations in the source code, so there is no point in worrying about them. You may wonder if sometimes you might make a better choice of register variable than the optimizer. The Institute has found with their own code that it’s very hard to do better than the optimizer. The optimizer has another (unfair) advantage. It can place a variable in a register for only part of its life time, leading to the elimination of some of the code from the caller.

Generally, the easiest way to do that is to copy the entire function to a header file and add the __inline keyword to it. Again, you need to choose which functions to do this for, and you’ll want to be sensitive to the time/space trade-off.

Of course, you’ll want to leave existing register declarations in portable code because they may be useful for other compilers, and the SAS/C Compiler will simply ignore them. You may even want to tailor register variable declarations towards other compilers that you use.

Avoid the SIZE and TIME Options

The SIZE and TIME options are not exactly what they might appear to be from their names. SIZE tells the optimizer not to attempt any optimizations that might increase code size. TIME tells the optim-
mizer not to attempt any optimizations that might increase program execution time. Thus, normally, you should not use these options. In most cases, you'd want the optimizer to do optimizations, even if there were some risk of increasing execution time (this will not usually occur). So it's best to leave these options out and let the optimizer do its best.

In most cases, you'd want the optimizer to do optimizations, even if there might sometimes be a space penalty (we're talking about a few bytes here). And you'd want the optimizer to do optimizations even if there were some risk of increasing execution time (this will usually occur). So it's best to leave these options out and let the optimizer do its best.

Only specify the SIZE option if code space is critical to your program, and you don't care about execution time. Only specify the SIZE option if execution time is critical to your program and you don't care about code space. And never specify the options together. Doing so can result in code that is both slower and larger than if you had specified neither.

Use the LOOP Option

The LOOP option tells the optimizer that it's okay to move safe code out of loops. By default, the optimizer does not move code out of loops because the code might then be executed when the loop had a zero iteration count, increasing execution time. However, it's rare that a loop is executed zero times, and even if it is, you may not mind the extra time that the moved code takes (at least no time is being spent in the loop). So your program will probably benefit from having code moved out of loops, and you should tell the optimizer that with the LOOP option.

Note that unsafe code, such as the example discussed above under floating-point arithmetic, will still not be moved out of loops. You'll need to move potentially unsafe code yourself. LOOP applies to safe code where movement might increase execution time.

A TECHNIQUE SPECIFIC TO THE SAS/C COMPILER FOR IBM MAINFRAMES

The following technique applies only to the SAS/C compiler for IBM 370 architecture mainframes. While this technique probably won't hurt optimizations on other machines, it probably won't help either.

Generating BXLE, BXH and BCT Instructions

The BXLE (branch on index low or equal), BXH (branch on index high), and BCT (branch on count) instructions are 370 instructions for controlling loops. Because these instructions combine an increment or decrement, a test, and a branch, it's normally better to generate them in place of the three or more instructions that would otherwise be needed. The IBM 3090 series hardware further optimizes BXLE and BXH, increasing their advantage.

Of course, you'd like the compiler to generate these instructions for every loop, but you need to provide some assistance. The following examples use for loops since for is the most commonly used form of C loop control. The basic form to generate a BXLE or BXH is:

```c
for (i = constant; i (comparison) value; i += increment)
```

`i` must be a signed integer or long (that is, not unsigned), or a pointer, because BXLE and BXH perform signed comparisons. Ideally the (comparison) operator should be <= (less than or equal) or > (greater than) since these are the types of comparisons the instructions perform. If you use the = or <= operators, then "value" has to be a constant (the optimizer can then change the constant and the operator to one supported by the instruction). You can use -= instead of += for the increment, but only if it's a constant (again the optimizer can change the constant value appropriately). The += and -= operators do not generate BXLE and BXH, so you should change code like that shown here:

```c
for (i = 100; i <= 1; i += 4)
```

```c
to
for (i = 99; i > 0; i -= 4)
```

The BCT rules are a little different because the instruction decrements by 1 only and tests for equality with zero. Thus the basic form is

```c
for (i = constant; i != 0; i--)
```

Here, `i` can be a signed or unsigned integer or long but not a pointer. Again, you'll want to re-code other forms, in particular

```c
for (i = constant; i > 0; i--)
```

Although this will generate a BXH, a BCT is a lot better because it uses fewer registers.

SAS/C is a registered trademark of SAS Institute Inc., Cary, NC, USA.

Lattice C Compiler is a trademark of Lattice, Inc.

Intel is a registered trademark of Intel Corporation.

IBM is a registered trademark of IBM Corp.