

ELIMINATING CLUTTER - SUBROUTINE CALLS*

Charles Severance
Kevin Dowd

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 3.0[†]

A typical corporation is full of frightening examples of overhead. Say your department has prepared a stack of paperwork to be completed by another department. What do you have to do to transfer that work? First, you have to be sure that your portion is completed; you can't ask them to take over if the materials they need aren't ready. Next, you need to package the materials — data, forms, charge numbers, and the like. And finally comes the official transfer. Upon receiving what you sent, the other department has to unpack it, do their job, repackage it, and send it back.

A lot of time gets wasted moving work between departments. Of course, if the overhead is minimal compared to the amount of useful work being done, it won't be that big a deal. But it might be more efficient for small jobs to stay within one department. The same is true of subroutine and function calls. If you only enter and exit modules once in a relative while, the overhead of saving registers and preparing argument lists won't be significant. However, if you are repeatedly calling a few small subroutines, the overhead can buoy them to the top of the profile. It might be better if the work stayed where it was, in the calling routine.

Additionally, subroutine calls inhibit compiler flexibility. Given the right opportunity, you'd like your compiler to have the freedom to intermix instructions that aren't dependent upon each other. These are found on either side of a subroutine call, in the caller and callee. But the opportunity is lost when the compiler can't peer into subroutines and functions. Instructions that might overlap very nicely have to stay on their respective sides of the artificial fence.

It helps if we illustrate the challenge that subroutine boundaries present with an exaggerated example. The following loop runs very well on a wide range of processors:

```
DO I=1,N
  A(I) = A(I) + B(I) * C
ENDDO
```

The code below performs the same calculations, but look at what we have done:

*Version 1.3: Aug 25, 2010 10:26 am -0500

[†]<http://creativecommons.org/licenses/by/3.0/>

```

DO I=1,N
  CALL MADD (A(I), B(I), C)
ENDDO
SUBROUTINE MADD (A,B,C)
  A = A + B * C
  RETURN
END

```

Each iteration calls a subroutine to do a small amount of work that was formerly within the loop. This is a particularly painful example because it involves floating-point calculations. The resulting loss of parallelism, coupled with the procedure call overhead, might produce code that runs 100 times slower. Remember, these operations are pipelined, and it takes a certain amount of “wind-up” time before the throughput reaches one operation per clock cycle. If there are few floating-point operations to perform between subroutine calls, the time spent winding up and winding down pipelines figures prominently.

Subroutine and function calls complicate the compiler’s ability to efficiently manage `COMMON` and `external` variables, delaying until the last possible moment actually storing them in memory. The compiler uses registers to hold the “live” values of many variables. When you make a call, the compiler cannot tell whether the subroutine will be changing variables that are declared as `external` or `COMMON`. Therefore, it’s forced to store any modified `external` or `COMMON` variables back into memory so that the callee can find them. Likewise, after the call has returned, the same variables have to be reloaded into registers because the compiler can no longer trust the old, register-resident copies. The penalty for saving and restoring variables can be substantial, especially if you are using lots of them. It can also be unwarranted if variables that ought to be local are specified as `external` or `COMMON`, as in the following code:

```

COMMON /USELESS/ K
DO K=1,1000
  IF (K .EQ. 1) CALL AUX
ENDDO

```

In this example, `K` has been declared as a `COMMON` variable. It is used only as a do-loop counter, so there really is no reason for it to be anything but local. However, because it is in a `COMMON` block, the call to `AUX` forces the compiler to store and reload `K` each iteration. This is because the side effects of the call are unknown.

So far, it looks as if we are preparing a case for huge main programs without any subroutines or functions! Not at all. Modularity is important for keeping source code compact and understandable. And frankly, the need for maintainability and modularity is always more important than the need for *small* performance improvements. However, there are a few approaches for streamlining subroutine calls that don’t require you to scrap modular coding techniques: macros and procedure inlining.

Remember, if the function or subroutine does a reasonable amount of work, procedure call overhead isn’t going to matter very much. However, if one small routine appears as a leaf node in one of the busiest sections of the call graph, you might want to think about inserting it in appropriate places in the program.

1 Macros

Macros are little procedures that are substituted inline at compile time. Unlike subroutines or functions, which are included once during the link, macros are replicated every place they are used. When the compiler

makes its first pass through your program, it looks for patterns that match previous macro definitions and expands them inline. In fact, in later stages, the compiler sees an expanded macro as source code.

Macros are part of both C and FORTRAN (although the FORTRAN notion of a macro, the *statement function*, is reviled by the FORTRAN community, and won't survive much longer).¹ For C programs, macros are created with a `#define` construct, as demonstrated here:

```
#define average(x,y) ((x+y)/2)
main ()
{
    float q = 100, p = 50;
    float a;
    a = average(p,q);
    printf ("%f\n",a);
}
```

The first compilation step for a C program is a pass through the C preprocessor, *cpp*. This happens automatically when you invoke the compiler. *cpp* expands `#define` statements inline, replacing the pattern matched by the macro definition. In the program above, the statement:

```
a = average(p,q);
```

gets replaced with:

```
a = ((p+q)/2);
```

You have to be careful how you define the macro because it literally replaces the pattern located by *cpp*. For instance, if the macro definition said:

```
#define multiply(a,b) (a*b)
```

and you invoked it as:

```
c = multiply(x+t,y+v);
```

¹The statement function has been eliminated in FORTRAN 90.

the resulting expansion would be $x+t*y+v$ — probably not what you intended.

If you are a C programmer you may be using macros without being conscious of it. Many C header files (*.h*) contain macro definitions. In fact, some “standard” C library functions are really defined as macros in the header files. For instance, the function *getchar* can be linked in when you build your program. If you have a statement:

```
#include <stdio.h>
```

in your file, *getchar* is replaced with a macro definition at compile time, replacing the C library function.

You can make *cpp* macros work for FORTRAN programs too.² For example, a FORTRAN version of the C program above might look like this:

```
#define AVERAG(X,Y) ((X+Y)/2)
C
  PROGRAM MAIN
  REAL A,P,Q
  DATA P,Q /50.,100./
  A = AVERAG(P,Q)
  WRITE (*,*) A
  END
```

Without a little preparation, the `#define` statement is rejected by the FORTRAN compiler. The program first has to be preprocessed through *cpp* to replace the use of `AVERAG` with its macro definition. It makes compilation a two-step procedure, but that shouldn’t be too much of a burden, especially if you are building your programs under the control of the *make* utility. We would also suggest you store FORTRAN programs containing *cpp* directives under *filename.F* to distinguish them from unadorned FORTRAN. Just be sure you make your changes only to the *.F* files and not to the output from *cpp*. This is how you would preprocess FORTRAN *.F* files by hand:

```
% /lib/cpp -P < average.F > average.f
% f77 average.f -c
```

The FORTRAN compiler never sees the original code. Instead, the macro definition is substituted inline as if you had typed it yourself:

```
C
  PROGRAM MAIN
```

²Some programmers use the standard UNIX *m4* preprocessor for FORTRAN

```
REAL A,P,Q
DATA P,Q /50.,100./ A = ((P+Q)/2)
WRITE (*,*) A
END
```

By the way, some FORTRAN compilers recognize the *.F* extension already, making the two-step process unnecessary. If the compiler sees the *.F* extension it invokes *cpp* automatically, compiles the output, and throws away the intermediate *.f* file. Try compiling a *.F* on your computer to see if it works.

Also, be aware that macro expansions may make source lines extend past column 72, which will probably make your FORTRAN compiler complain (or worse: it might pass unnoticed). Some compilers support input lines longer than 72 characters. On the Sun compilers the `-e` option allows extended input lines up to 132 characters long.

2 Procedure Inlining

Macro definitions tend to be pretty short, usually just a single statement. Sometimes you have slightly longer (but not too long) bits of code that might also benefit from being copied inline, rather than called as a subroutine or function. Again, the reason for doing this is to eliminate procedure call overhead and expose parallelism. If your compiler is capable of *inlining* subroutine and function definitions into the modules that call them, then you have a very natural, very portable way to write modular code without suffering the cost of subroutine calls.

Depending on the vendor, you can ask the compiler for procedure inlining by:

- Specifying which routines should be inlined on the compiler's command line
- Putting inlining directives into the source program
- Letting the compiler inline automatically

The directives and compile line options are not standard, so you have to check your compiler documentation. Unfortunately, you may learn that there is no such feature ("yet," always yet), or that it's an expensive extra. The third form of inlining in the list, automatic, is available from just a few vendors. Automatic inlining depends on a sophisticated compiler that can view the definitions of several modules at once.

There are some words of caution with regard to procedure inlining. You can easily do too much of it. If everything and anything is ingested into the body of its parents, the resulting executable may be so large that it repeatedly spills out of the instruction cache and becomes a net performance loss. Our advice is that you use the caller/callee information profilers give you and make some intelligent decisions about inlining, rather than trying to inline every subroutine available. Again, small routines that are called often are generally the best candidates for inlining.