Eliminating Clutter - Other Clutter*

Charles Severance
Kevin Dowd

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

Clutter comes in many forms. Consider the previous sections as having dealt with large pieces of junk you might find in the front hall closet: an ironing board, hockey sticks, and pool cues. Now we are down to the little things: a widowed checker, a tennis ball, and a hat nobody owns. We want to mention a few of them here. We apologize in advance for changing subjects a lot, but that's the nature of cleaning out a closet!

1 Data Type Conversions

Statements that contain runtime type conversions suffer a little performance penalty each time the statement is executed. If the statement is located in a portion of the program where there is a lot of activity, the total penalty can be significant.

People have their reasons for writing applications with mixed typing. Often it is a matter of saving memory space, memory bandwidth, or time. In the past, for instance, double-precision calculations took twice as long as their single-precision counterparts, so if some of the calculations could be arranged to take place in single precision, there could be a performance win.[1] But any time saved by performing part of the calculations in single precision and part in double precision has to be measured against the additional overhead caused by the runtime type conversions. In the following code, the addition of `A(I)` to `B(I)` is *mixed type*:

```
INTEGER NUMEL, I
PARAMETER (NUMEL = 1000)
REAL*8 A(NUMEL)
REAL*4 B(NUMEL)
DO I=1,NUMEL
  A(I) = A(I) + B(I)
ENDDO
```

*Version 1.3: Aug 25, 2010 10:26 am -0500
†http://creativecommons.org/licenses/by/3.0/
[1] Nowadays, single-precision calculations can take *longer* than double-precision calculations from register to register.

In each iteration, `B(I)` has to be promoted to double precision before the addition can occur. You don't see the promotion in the source code, but it's there, and it takes time.

C programmers beware: in Kernighan and Ritchie (K&R) C, all floating-point calculations in C programs take place in double precision — even if all the variables involved are declared as *float*. It is possible for you to write a whole K+R application in one precision, yet suffer the penalty of many type conversions.

Another data type–related mistake is to use character operations in IF tests. On many systems, character operations have poorer performance than integer operations since they may be done via procedure calls. Also, the optimizers may not look at code using character variables as a good candidate for optimization. For example, the following code:

```
DO I=1,10000
  IF ( CHVAR(I) .EQ. 'Y' ) THEN
    A(I) = A(I) + B(I)*C
  ENDIF
ENDDO
```

might be better written using an integer variable to indicate whether or not a computation should be performed:

```
DO I=1,10000
  IF ( IFLAG(I) .EQ. 1 ) THEN
    A(I) = A(I) + B(I)*C
  ENDIF
ENDDO
```

Another way to write the code, assuming the IFLAG variable was 0 or 1, would be as follows:

```
DO I=1,10000
  A(I) = A(I) + B(I)*C*IFLAG(I)
ENDDO
```

The last approach might actually perform slower on some computer systems than the approach using the IF and the integer variable.

2 Doing Your Own Common Subexpression Elimination

So far we have given your compiler the benefit of the doubt. *Common subexpression elimination* — the ability of the compiler to recognize repeated patterns in the code and replace all but one with a temporary variable — probably works on your machine for simple expressions. In the following lines of code, most compilers would recognize `a+b` as a common subexpression:

```
c = a + b + d
e = q + a + b
```

becomes:

```
temp = a + b
c = temp + d
e = q + temp
```

Substituting for `a+b` eliminates some of the arithmetic. If the expression is reused many times, the savings can be significant. However, a compiler's ability to recognize common subexpressions is limited, especially when there are multiple components, or their order is permuted. A compiler might not recognize that `a+b+c` and `c+b+a` are equivalent.[2] For important parts of the program, you might consider doing common subexpression elimination of complicated expressions by hand. This guarantees that it gets done. It compromises beauty somewhat, but there are some situations where it is worth it.

Here's another example in which the function sin is called twice with the same argument:

```
x = r*sin(a)*cos(b);
y = r*sin(a)*sin(b);
z = r*cos(a);
```

becomes:

```
temp = r*sin(a);
x = temp*cos(b);
y = temp*sin(b);
z = r*cos(a);
```

We have replaced one of the calls with a temporary variable. We agree, the savings for eliminating one transcendental function call out of five won't win you a Nobel prize, but it does call attention to an important point: compilers typically do not perform common subexpression elimination over subroutine or function calls. The compiler can't be sure that the subroutine call doesn't change the state of the argument or some other variables that it can't see.

The only time a compiler might eliminate common subexpressions containing function calls is when they are intrinsics, as in FORTRAN. This can be done because the compiler can assume some things about their side effects. You, on the other hand, can see into subroutines, which means you are better qualified than the compiler to group together common subexpressions involving subroutines or functions.

[2] And because of overflow and round-off errors in floating-point, in some situations they might not be equivalent.

3 Doing Your Own Code Motion

All of these optimizations have their biggest payback within loops because that's where all of a program's activity is concentrated. One of the best ways to cut down on runtime is to move unnecessary or repeated (invariant) instructions out of the main flow of the code and into the suburbs. For loops, it's called *hoisting* instructions when they are pulled out from the top and *sinking* when they are pushed down below. Here's an example:

```
DO I=1,N
  A(I) = A(I) / SQRT(X*X + Y*Y)
ENDDO
```

becomes:

```
TEMP = 1 / SQRT(X*X + Y*Y)
DO I=1,N
  A(I) = A(I) * TEMP
ENDDO
```

We hoisted an expensive, invariant operation out of the loop and assigned the result to a temporary variable. Notice, too, that we made an algebraic simplification when we exchanged a division for multiplication by an inverse. The multiplication will execute much more quickly. Your compiler might be smart enough to make these transformations itself, assuming you have instructed the compiler that these are legal transformations; but without crawling through the assembly language, you can't be positive. Of course, if you rearrange code by hand and the runtime for the loop suddenly goes down, you will know that the compiler has been sandbagging all along.

Sometimes you want to sink an operation below the loop. Usually, it's some calculation performed each iteration but whose result is only needed for the last. To illustrate, here's a sort of loop that is different from the ones we have been looking at. It searches for the final character in a character string:

```
while (*p != ' ')
  c = *p++;
```

becomes:

```
while (*p++ != ' ');
c = *(p-1);
```

The new version of the loop moves the assignment of `c` beyond the last iteration. Admittedly, this transformation would be a reach for a compiler and the savings wouldn't even be that great. But it illustrates the notion of sinking an operation very well.

Again, hoisting or sinking instructions to get them out of loops is something your compiler should be capable of doing. But often you can slightly restructure the calculations yourself when you move them to get an even greater benefit.

4 Handling Array Elements in Loops

Here's another area where you would like to trust the compiler to do the right thing. When making repeated use of an array element within a loop, you want to be charged just once for loading it from memory. Take the following loop as an example. It reuses `X(I)` twice:

```
DO I=1,N
  XOLD(I) = X(I)
  X(I)= X(I) + XINC(I)
ENDDO
```

In reality, the steps that go into retrieving `X(I)` are just additional common subex- pressions: an address calculation (possibly) and a memory load operation. You can see that the operation is repeated by rewriting the loop slightly:

```
DO I=1,N
  TEMP= X(I)
  XOLD(I) = TEMP
  X(I)= TEMP + XINC(I)
ENDDO
```

FORTRAN compilers *should* recognize that the same `X(I)` is being used twice and that it only needs to be loaded once, but compilers aren't always so smart. You sometimes have to create a temporary scalar variable to hold the value of an array element over the body of a loop. This is particularly true when there are subroutine calls or functions in the loop, or when some of the variables are `external` or `COMMON`. Make sure to match the types between the temporary variables and the other variables. You don't want to incur type conversion overhead just because you are "helping" the compiler. For C compilers, the same kind of indexed expres- sions are an even greater challenge. Consider this code:

```
doinc(int xold[],int x[],int xinc[],int n)
{
  for (i=0; i<n; i++) {
    xold[i] = x[i];
    x[i]= x[i] + xinc[i];
  }
```

```
}
```

Unless the compiler can see the definitions of `x`, `xinc`, and `xold`, it has to assume that they are pointers leading back to the same storage, and repeat the loads and stores. In this case, introducing temporary variables to hold the values `x`, `xinc`, and `xold` is an optimization the compiler wasn't free to make.

Interestingly, while putting scalar temporaries in the loop is useful for RISC and superscalar machines, it doesn't help code that runs on parallel hardware. A parallel compiler looks for opportunities to eliminate the scalars or, at the very least, to replace them with temporary vectors. If you run your code on a parallel machine from time to time, you might want to be careful about introducing scalar temporary variables into a loop. A dubious performance gain in one instance could be a real performance loss in another.