Guidelines for writing efficient C/C++ code

Greg Davis, Green Hills Software, Inc.

APRIL 05, 2006

Simple source code changes can often result in substantial performance enhancements using modern optimizing compilers on high-end embedded processors. But, why is performance necessary? After all, the capabilities of modern microprocessors dwarf the capabilities of 1980-era supercomputers.

First, the average case response time of a real-time system is irrelevant. It is the worst-case response time that guards against a dropped call, a misprint, or other error conditions in a product. In other words, performance is necessary to minimize the response time of one's system, thereby achieving product reliability.

Second, increased performance allows for the implementation of more features without compromising the integrity of the system. Conversely, performance might be used to select a cheaper microprocessor or one that consumes less power.

There are many factors that determine the performance of a system. The choice of hardware can mean the difference between a few MIPS and a few hundred. Good data structures and algorithms are essential, and bookshelves have been written on this topic. A good compiler is also essential. One should evaluate the features and optimization capabilities of a compiler before spending too much time working with it.

The purpose here is to explore an often-overlooked aspect of achieving maximum performance. No matter what hardware is chosen, which data structures and algorithms are employed, and which compiler is used, proper coding guidelines can dramatically impact the efficiency of one's code.

Nonetheless, developers are often unaware of the consequences of their programming habits. And unlike fundamental design decisions, many of these improvements can be made at a late stage of a project.

In the following examples, efficient coding guidelines will be illustrated using the C and C++ programming languages and, at times, PowerPC, ARM, and x86 assembly code. Many of these concepts, however, apply to other programming languages. Most of them are also processor independent, although there are some exceptions, which will be noted later.

Choice of Data Type Sizes

The most fundamental data type in the C language is the integer, which is normally defined as the most efficient type for a given processor. The C language treats this type specially in that the language does not operate on smaller types, at least conceptually speaking.

If a character is incremented, the language specifies that the character is first promoted to an integer, the addition is performed as an integer, and the truncated result is stored back into the character.

In practice, using a smaller type for a local variable is computationally inefficient. Consider the following snippet of code:

```
int m;
char c;
m += ++c;
```

In PowerPC assembly code, this looks like:

On a CISC chip like the 80386, it looks like:

On a RISC chip, one pays the price after writing to a sub-integral variable " the value must be sign or zero extended in register to ensure that the high bits of the register are consistently set. On a CISC chip, one pays the price when mixing the short variable with a larger integer variable. And unless you're using an 8 or 16-bit microprocessor, the microprocessor does not save on the shorter add instruction, so there is nothing to make up for the cost of the sign or zero extension after the increment.
Variables in memory are sometimes different. Because memory is scarce, it often makes sense to conserve by using a smaller data type. Most microprocessors can load and extend the short value in a single instruction, which means that the extension is free. Also, the truncation that is necessary when storing the value can often be avoided because sub-integral store instructions generally ignore the high bits.

These choices can be simplified by using the C99 header **stdint.h**. Even if your compiler does not support C99, it might provide stdint.h, or you may write it yourself. This file defines types that fit into a few categories:

1) i**nt***size***_t** " Used when a type must be exactly a given length.
2) int_least*size***_t** " Used when a type must be at least a given size and when the type should be optimized for space. This is preferred for data structures.
3) int_fast*size***_t** " Used when a type must be at least a given size and when the type should be as fast as possible. This is preferred for local variables.

Larger data types should also be used sparingly. For example, C99 adds the long long type. This type has been available as an extension to C89 in most compilers for years. On a 32-bit microprocessor, these variables take up two registers and require multiple operations to perform arithmetic on them. An addition or a subtraction can usually be done inline in a few instructions.
However, an innocuous looking shift can turn into quite a few instructions. And you can forget about division! Of course, these variables might be necessary or convenient when coding, but do not use them unless you need them and understand the corresponding code size and speed impact.

Variable signedness
Unsigned variables can result in more efficient generated code for certain operations than using signed variables. For example, unsigned division by a power of two can be performed as a right shift.

Most architectures require a few instructions to perform a signed divide by a power of two. Likewise, computing an unsigned modulus by a power of two can be implemented as an AND operation (or a bit extract). Computing a signed modulus is more complicated. In addition, it is sometimes useful that unsigned variables can represent just over twice as many positive values as their signed counterparts.

Characters deserve special mention here. Many architectures, such as the ARM, and PowerPC, are more efficient at loading unsigned characters than they are at loading signed characters. Other architectures, such as the V800 and SH, handle signed characters more efficiently. Most architectures' ABIs define the plain "char" type as whichever type is most efficient. So, unless a character needs to be signed or unsigned, use the plain char type.

Use of access types
For global data, use the **static** keyword (or C++ anonymous namespaces) whenever possible. In some cases, **static** allows a compiler to deduce things about the access patterns to a variable. The **static** keyword also "hides" the data, which is generally a good thing from a programming practices standpoint. Declaring a function as **static** is also helpful in many cases.
Example:

This code can be optimized by the compiler in a couple of ways because the functions were declared as **static**. First, the compiler can inline the call to *allocate_new_unique_number()* and delete the out-of-line copy, because it is not called from any other place.

Second, if the compiler has basic information on the external *allocate_other_fields()*function, the compiler can sometimes tell that the function will not call back into this module. This knowledge allows eliminating the second load of unique_number embedded in the inlined *allocate_new_unique_number()* function.

<>On the other hand, **static** data should be avoided whenever possible for function-local data. The data's value must be preserved between calls to the function, making it very expensive to access the data, and requiring permanent RAM storage. The static keyword should only be used on function-local data when the value must be preserved across calls or for large data allocations (such as an array) when the programmer prefers to tradeoff consumption of stack space for permanent RAM.

Global Variable Definition

Compilers can sometimes optimize accesses to global variables if they are defined and used together in the same module. In that case, the compiler can access one variable as an offset from another variable's address, as if they were both members of the same structure. Therefore, all other things being equal, it is worthwhile to define global variables in the modules where they are used the most.

For example, if the global variable "**glob**" is used the most in file.c, it should be defined there. Use a definition such as:

int glob;
or:
int glob = 1;

in addition to declaring glob (with **extern int glob;**) in a header file so that other modules can reference it.

Some programmers get confused about uninitialized variable definitions, so it is worth clarifying how they are often implemented. Most C compilers support a feature, called common variables, where uninitialized global variable definitions are combined and resolved at link-time. For example, a user could place the definition:

int glob;

in a header file and include this header file in every file in the project. While a strict reading of the C language implies that this will result in link-time errors, under a common variable model, each module will output a special common reference, which is different from a traditional definition.

The linker will combine all of the common references for a variable, and if the variable was not already defined elsewhere, allocate space for the variable in an uninitialized data section (such as **.bss**) and point all of the common references to this newly allocated space. On the other hand, if the user employs a definition such as:

int glob = 1;

in one module, all other uninitialized, common references would resolve to this definition.

It is best to write code without defining the same uninitialized global variable in multiple modules. Then, if you turn off the common variable feature in the compiler, the compiler is able to perform more aggressive optimizations because it knows that uninitialized global variable definitions are true definitions.

Volatile Variables

The volatile keyword tells the compiler not to optimize away accesses to the variable or type. That is, the value will be loaded each time it is needed, and will be stored each time it is modified. Volatile variables have two major uses:

1) The variable is a memory mapped hardware device where the value can change asynchronously and/or where it is critical that all writes to the variable are respected.
2) The variable is used in a multi-threading environment where another thread may modify the variable at any time.

h

The alternative to careful use of the volatile keyword is to disable so-called "memory optimizations". Effectively, all variables are treated as volatile when this option is chosen. Because disabling memory optimizations are important for efficient code, developers are encouraged to choose a less conservative approach.

Separate threads often perform separate functions. As a result, they may have many variables and data structures that are not accessed by other tasks. Good software engineering practices might be able to minimize the overlap to a few shared header files. If such practices have reduced the scope of the code to a few files, it is more feasible to find the variables and data structures that are shared between threads.

Const
The const keyword is helpful in a couple of ways. First, const variables can usually be allocated to a read-only data section, which can save on the amount of RAM required if the read-only data is allocated to flash or ROM. Second, the const keyword, when applied to a pointer or reference parameter, might allow the compiler to deduce that the call will not result in the value being modified.

Restrict
The restrict keyword tells a compiler that the specified pointer is the only way to access a given piece of data. This can allow the compiler to optimize more aggressively. Consider the following example, a simple finite impulse response code:

Consider the inner loop for a PowerPC target:

Better code can be generated by pulling the first load out of the loop, since **in[i]** does not change within the inner loop. However, the compiler cannot tell that **in[i]** does not change within the loop. The compiler is unable to determine this because the in and out arrays could overlap. If the function declaration is changed to:

void fir_and_copy(int *in, const int *coeff, int *restrict out)

the compiler knows that writes through ***out** cannot change the values in ***in**. The restrict keyword is a bit confusing because it applies to the pointer itself rather than the data the pointer points to (contrast const int ***x** and **int *restrict x**).

Pointers and the & operator
It is usually more efficient to have a function return a scalar value than to pass the scalar value by reference or by address. For example, often times a value is returned from a function by passing the address of an integer to the function. For example:

Taking the address of a variable forces the compiler to allocate the variable on the stack, all but assuring less efficient code generation. Passing an argument as a C++ reference parameter has the same effect.

Declaration scope of variables
Declare a variable in the inner-most scope possible, particularly when its address must be taken. In such cases, the compiler must keep the variable around on the stack until it goes out of scope. This can inhibit certain optimizations that depend on the variable being dead. For example, consider the variable "loc" in the following function:

The compiler could potentially perform a tail call for the call to *func2()*. This is an optimization where the frame for *func()*, if any, is popped, and the last call instruction to *func2()* is replaced by a branch to func2().

This saves the need to return to *func1()*, which would immediately return to its caller. Instead, *func2()* returns directly to *func1()*'s caller. Unfortunately, the compiler cannot employ this optimization because it cannot determine that *loc* is not used in the second call to *func2()* (which is possible if its address was saved in the first call). The following code allows for better optimization:

In this case, the compiler knows that the lifetime of *loc* ends before the final call " and the tail call, at least in principle, can happen. Another benefit of using inner scopes is that variables from non-overlapping scopes can share the same space on the stack. This helps to minimize stack use and can result in smaller code size on some architectures.

However, it is usually not worthwhile to create artificially small scopes simply to bound the lifetimes of variables.

Floating Point Arithmetic
Understanding the rules of arithmetic promotion can help you avoid costly mistakes. Many embedded architectures do not implement floating point arithmetic in hardware. Some processors implement the single precision "**float**" type in hardware, but leave the "**double**" type to floating point software emulation routines.
Unless doubles are implemented in hardware, it is more efficient to do arithmetic with the single precision type. If this is the case and if single precision arithmetic is sufficient, follow these rules:

1. Write single precision floating point constants with the **F** suffix. For example, write **3.0F**instead of **3.0**. The constant **3.0** is a double precision value, which forces the surrounding arithmetic expressions to be promoted to double precision.
2. Use single precision math routines, such as **sinf()** instead of **sin()**, the double precision version.
3. Avoid old-style prototypes and function declarations because these force floats to be promoted to double. So, instead of:

float foo(f)
float f;

do:

float foo(float f);

This is probably only a concern for old code bases.

Variable Length Arrays
The variable length array feature, which is included in C99, might result in less efficient array access. In cases where the array is multi-dimensional and subscripts other than the first are of variable lengths, the resulting code may be larger and slower. The feature is useful, but be aware that code generation can suffer.

Low Level Assembly Instructions
Sometimes it is helpful or necessary to use specific assembly instructions in embedded programming. Intrinsic functions are the best way to do this. Intrinsic functions look like function calls, but they are inlined into specific assembly instructions. Refer to your compiler vendor's documentation to determine which intrinsics are available.

Inlined assembly code uses non-portable syntax and compilers generally make over-conservative assumptions when encountering inlined assembly, thus affecting code performance. Intrinsics can be #define'd into other names if switching from one compiler to another. This can be done once in a header file rather than going through the code to see every place where inlined assembly was used.

For example, instead of writing the following code to disable interrupts on the PowerPC:

Manual Loop Tricks

Sometimes programmers feel compelled to manually unroll loops, perform strength reduction, or use other transformations that a standard compiler optimizer would handle. For example:

is sometimes manually transformed into:

Such transformations are usually only effective under fairly specific architectures and compilers. For example, the 32-bit ARM architecture supports the post-increment addressing mode used above, but the PowerPC architecture only includes the pre-increment addressing mode. So, for the PowerPC, this loop could be written as:

As a general rule, only do manual transformations for time critical sections of your code where your compiler of choice has not been able to perform adequate optimizations on its own, even after adjusting compilation options. Write simple code for most cases and let the compiler do the optimization work for you.

Conclusion

The performance impact of some decisions that programmers make when writing their code can be significant. While efficient algorithmic design is of the highest importance, making intelligent choices when implementing the design can help application code perform at its highest potential.