Writing Efficient C and C Code Optimization

Koushik Ghosh, 26 Feb 2004

4.72 (78 votes)

Rate this: vote 1vote 2vote 3vote 4vote 5

In this article, I have gathered all the experiences and information, which can be applied to make a C code optimized for speed as well as memory.

Introduction

During a project for developing a light JPEG library which is enough to run on a mobile device without compromising quality graphics on a mobile device, I have seen and worked out a number of ways in which a given computer program can be made to run faster. In this article, I have gathered all the experiences and information, which can be applied to make a C code optimized for speed as well as memory.

Although a number of guidelines are available for C code optimization, there is no substitute for having a thorough knowledge of the compiler and machine for which you are programming.

Often, speeding up a program can also cause the code's size to increase. This increment in code size can also have an adverse effect on a program's complexity and readability. It will not be acceptable if you are programming for small device like mobiles, PDAs etc., which have strict memory restrictions. So, during optimization, our motto should be to write the code in such a way that memory and speed both will be optimized.

Declaration

Actually, during my project, I have used the tips from this for optimization ARM because my project was on ARM platform, but I have also used many other articles from Internet. All tips of every article do not work well, so I collect only those tips together, which are very useful and very efficient. Also, I have modified some of them in such a way that they are almost applicable for all the environments apart from ARM.

What I did is just make a collection of the information from various sites but mostly from that PDF file I mentioned above. I never claimed that these are my own discoveries. I have mentioned all information sources in the*References* section at the end of this article.

Where it is needed?

Without this point, no discussion can be started. First and the most important part of optimizing a computer program is to find out where to optimize, which portion or which module of the program is running slow or using huge memory. If each part is separately being optimized then the total program will be automatically faster.

The optimizations should be done on those parts of the program that are run the most, especially those methods which are called repeatedly by various inner loops that the program can have.

For an experienced programmer, it will usually be quite easy to find out the portions where a program requires the most optimization attention. But there are a lot of tools also available for detecting those parts of a program. I have used Visual C ++ IDE's in-built profiler to find out where the program spends most click tricks. Another tool I have used is Intel Vtune, which is a very good profiler for detecting the slowest parts of a program. In my experience, it will usually be a particular inner or nested loop, or a call to some third party library methods, which is the main culprit for running the program slow.

Integers

We should use `unsigned int` instead of `int` if we know the value will never be negative. Some processors can handle unsigned integer arithmetic considerably faster than signed (this is also good practice, and helps make for self-documenting code).

So, the best declaration for an `int` variable in a tight loop would be:

Hide Copy Code

```
register unsigned int variable_name;
```

although, it is not guaranteed that the compiler will take any notice of `register`, and `unsigned` may make no difference to the processor. But it may not be applicable for all compilers.

Remember, integer arithmetic is much faster than floating-point arithmetic, as it can usually be done directly by the processor, rather than relying on external FPUs or floating point math libraries.

We need to be accurate to two decimal places (e.g. in a simple accounting package), scale everything up by 100, and convert it back to floating point as late as possible.

Division and Remainder

In standard processors, depending on the numerator and denominator, a 32 bit division takes 20-140 cycles to execute. The division function takes a constant time plus a time for each bit to divide.

Hide Copy Code

```
Time (numerator / denominator) = C0 + C1* log2 (numerator / denominator)
     = C0 + C1 * (log2 (numerator) - log2 (denominator)).
```

The current version takes about 20 + 4.3N cycles for an ARM processor. As an expensive operation, it is desirable to avoid it where possible. Sometimes, such expressions can be rewritten by replacing the division by a multiplication. For example, `(a / b) > c` can be rewritten as `a > (c * b)` if it is known that `b` is positive and `b *c` fits in an integer. It will be better to use unsigned division by ensuring that one of the operands is unsigned, as this is faster than signed division.

Combining division and remainder

Both dividend (`x / y`) and remainder (`x % y`) are needed in some cases. In such cases, the compiler can combine both by calling the division function once because as it always returns both dividend and remainder. If both are needed, we can write them together like this example:

Hide Copy Code

```
int func_div_and_mod (int a, int b) {
       return (a / b) + (a % b);
   }
```

Division and remainder by powers of two

We can make a division more optimized if the divisor in a division operation is a power of two. The compiler uses a shift to perform the division. Therefore, we should always arrange, where possible, for scaling factors to be powers of two (for example, 64 rather than 66). And if it is unsigned, then it will be more faster than the signed division.

Hide Copy Code

```
typedef unsigned int uint;

    uint div32u (uint a) {
     return a / 32;
    }
    int div32s (int a){
     return a / 32;
    }
```

Both divisions will avoid calling the division function and the unsigned division will take fewer instructions than the signed division. The signed division will take more time to execute because it rounds towards zero, while a shift rounds towards minus infinity.

An alternative for modulo arithmetic

We use remainder operator to provide modulo arithmetic. But it is sometimes possible to rewrite the code using`if` statement checks.

Consider the following two examples:

Hide Copy Code

```
    uint modulo_func1 (uint count)
    {
       return (++count % 60);
    }

    uint modulo_func2 (uint count)
    {
       if (++count >= 60)
      count = 0;
      return (count);
    }
```

The use of the `if` statement, rather than the remainder operator, is preferable, as it produces much faster code. Note that the new version only works if it is known that the range of count on input is 0-59.

Using array indices

If you wished to set a variable to a particular character, depending upon the value of something, you might do this:

Hide Copy Code

```
switch ( queue ) {
case 0 :   letter = 'W';
   break;
case 1 :   letter = 'S';
   break;
case 2 :   letter = 'U';
   break;
}
```

Or maybe:

Hide Copy Code

```
if ( queue == 0 )
  letter = 'W';
else if ( queue == 1 )
  letter = 'S';
else
  letter = 'U';
```

A neater (and quicker) method is to simply use the value as an index into a character array, e.g.:

Hide Copy Code

```
static char *classes="WSU";

letter = classes[queue];
```

Global variables

Global variables are never allocated to registers. Global variables can be changed by assigning them indirectly using a pointer, or by a function call. Hence, the compiler cannot cache the value of a global variable in a register, resulting in extra (often unnecessary) loads and stores when globals are used. We should therefore not use global variables inside critical loops.

If a function uses global variables heavily, it is beneficial to copy those global variables into local variables so that they can be assigned to registers. This is possible only if those global variables are not used by any of the functions which are called.

For example:

Hide Copy Code

```
int f(void);
int g(void);
int errs;
void test1(void)
{
  errs += f();
  errs += g();
}

void test2(void)
{
  int localerrs = errs;
  localerrs += f();
  localerrs += g();
  errs = localerrs;
}
```

Note that `test1` must load and store the global `errs` value each time it is incremented, whereas `test2` stores`localerrs` in a register and needs only a single instruction.

Using Aliases

Consider the following example -

Hide Copy Code

```
void func1( int *data )
{
    int i;

    for(i=0; i<10; i++)
    {
          anyfunc( *data, i);
    }
}
```

Even though `*data` may never change, the compiler does not know that `anyfunc ()` did not alter it, and so the program must read it from memory each time it is used - it may be an alias for some other variable that is altered elsewhere. If we know it won't be altered, we could code it like this instead:

Hide Copy Code

```
void func1( int *data )
{
    int i;
    int localdata;

    localdata = *data;
    for(i=0; i<10; i++)
    {
         anyfunc ( localdata, i);
    }
}
```

This gives the compiler better opportunity for optimization.

Live variables and spilling

As any processor has a fixed set of registers, there is a limit to the number of variables that can be kept in registers at any one point in the program.

Some compilers support live-range splitting, where a variable can be allocated to different registers as well as to memory in different parts of the function. The live-range of a variable is defined as all statements between the last assignment to the variable, and the last usage of the variable before the next assignment. In this range, the value of the variable is valid, thus it is alive. In between live ranges, the value of a variable is not needed: it is dead, so its register can be used for other variables, allowing the compiler to allocate more variables to registers.

The number of registers needed for register-allocatable variables is at least the number of overlapping live-ranges at each point in a function. If this exceeds the number of registers available, some variables must be stored to memory temporarily. This process is called spilling.

The compiler spills the least frequently used variables first, so as to minimize the cost of spilling. Spilling of variables can be avoided by:

- Limiting the maximum number of live variables: this is typically achieved by keeping expressions simple and small, and not using too many variables in a function. Subdividing large functions into smaller, simpler ones might also help.
- Using register for frequently-used variables: this tells the compiler that the register variable is going to be frequently used, so it should be allocated to a register with a very high priority. However, such a variable may still be spilled in some circumstances.

Variable Types

The C compilers support the basic types `char`, `short`, `int` and `long` (signed and unsigned), `float` and `double`. Using the most appropriate type for variables is very important, as it can reduce code and data size and increase performance considerably.

Local variables

Where possible, it is best to avoid using `char` and `short` as local variables. For the types `char` and `short`, the compiler needs to reduce the size of the local variable to 8 or 16 bits after each assignment. This is called sign-extending for signed variables and zero extending for unsigned variables. It is implemented by shifting the register left by 24 or 16 bits, followed by a signed or unsigned shift right by the same amount, taking two instructions (zero-extension of an unsigned `char` takes one instruction).

These shifts can be avoided by using `int` and `unsigned int` for local variables. This is particularly important for calculations which first load data into local variables and then process the data inside the local variables. Even if data is input and output as 8- or 16-bit quantities, it is worth considering processing them as 32-bit quantities.

Consider the following three example functions:

Hide Copy Code

```
int wordinc (int a)
{
   return a + 1;
}
short shortinc (short a)
{
    return a + 1;
}
char charinc (char a)
{
    return a + 1;
}
```

The results will be identical, but the first code segment will run faster than others.

Pointers

If possible, we should pass structures by reference, that is pass a pointer to the structure, otherwise the whole thing will be copied onto the stack and passed, which will slow things down. I've seen programs that pass structures several Kilo Bytes in size by value, when a simple pointer will do the same thing.

Functions receiving pointers to structures as arguments should declare them as pointer to constant if the function is not going to alter the contents of the structure. As an example:

Hide Copy Code

```
void print_data_of_a_structure ( const Thestruct  *data_pointer)
{
    ...printf contents of the structure...
}
```

This example informs the compiler that the function does not alter the contents (as it is using a pointer to constant structure) of the external structure, and does not need to keep re-reading the contents each time they are accessed. It also ensures that the compiler will trap any accidental attempts by your code to write to the read-only structure and give an additional protection to the content of the structure.

Pointer chains

Pointer chains are frequently used to access information in structures. For example, a common code sequence is:

Hide Copy Code

```
typedef struct { int x, y, z; } Point3;
typedef struct { Point3 *pos, *direction; } Object;

void InitPos1(Object *p)
{
   p->pos->x = 0;
   p->pos->y = 0;
   p->pos->z = 0;
}
```

However, this code must reload `p->pos` for each assignment, because the compiler does not know that `p->pos->x` is not an alias for `p->pos`. A better version would cache `p->pos` in a local variable:

Hide Copy Code

```
void InitPos2(Object *p)
```

```
{
   Point3 *pos = p->pos;
   pos->x = 0;
   pos->y = 0;
   pos->z = 0;
}
```

Another possibility is to include the `Point3` structure in the `Object` structure, thereby avoiding pointers completely.

Conditional Execution

Conditional execution is applied mostly in the body of `if` statements, but it is also used while evaluating complex expressions with relational (`<`, `==`, `>` and so on) or boolean operators (`&&`, `!`, and so on). Conditional execution is disabled for code sequences which contain function calls, as on function return the flags are destroyed.

It is therefore beneficial to keep the bodies of `if` and `else` statements as simple as possible, so that they can be conditionalized. Relational expressions should be grouped into blocks of similar conditions.

The following example shows how the compiler uses conditional execution:

Hide Copy Code

```
int g(int a, int b, int c, int d)
{
   if (a > 0 && b > 0 && c < 0 && d < 0)
   //  grouped conditions tied up together//
      return a + b + c + d;
   return -1;
}
```

As the conditions were grouped, the compiler was able to conditionalize them.

Boolean Expressions & Range checking

A common boolean expression is used to check whether a variable lies within a certain range, for example, to check whether a graphics co-ordinate lies within a window:

Hide Copy Code

```
bool PointInRectangelArea (Point p, Rectangle *r)
```

```
{
   return (p.x >= r->xmin && p.x < r->xmax &&
                   p.y >= r->ymin && p.y < r->ymax);
}
```

There is a faster way to implement this: `(x >= min && x < max)` can be transformed into `(unsigned)(x-min) < (max-min)`. This is especially beneficial if `min` is zero. The same example after this optimization:

Hide Copy Code

```
bool PointInRectangelArea (Point p, Rectangle *r)
{
    return ((unsigned) (p.x - r->xmin) < r->xmax &&
   (unsigned) (p.y - r->ymin) < r->ymax);

}
```

Boolean Expressions & Compares with zero

The Processor flags are set after a compare (i.e. `CMP`) instruction. The flags can also be set by other operations, such as `MOV`, `ADD`, `AND`, `MUL`, which are the basic arithmetic and logical instructions (the data processing instructions). If a data processing instruction sets the flags, the `N` and `Z` flags are set the same way as if the result was compared with zero. The `N` flag indicates whether the result is negative, the `Z` flag indicates that the result is zero.

The `N` and `Z` flags on the processor correspond to the signed relational operators `x < 0`, `x >= 0`, `x == 0`, `x !=0`, and unsigned `x == 0`, `x != 0` (or `x > 0`) in C.

Each time a relational operator is used in C, the compiler emits a compare instruction. If the operator is one of the above, the compiler can remove the compare if a data processing operation preceded the compare. For example:

Hide Copy Code

```
int aFunction(int x, int y)
{
   if (x + y < 0)
      return 1;
  else
     return 0;
```

```
}
```

If possible, arrange for critical routines to test the above conditions. This often allows you to save compares in critical loops, leading to reduced code size and increased performance. The C language has no concept of a carry flag or overflow flag, so it is not possible to test the `C` or `V` flag bits directly without using inline assembler. However, the compiler supports the carry flag (unsigned overflow). For example:

Hide Copy Code

```
int sum(int x, int y)
{
   int res;
   res = x + y;
   if ((unsigned) res < (unsigned) x) // carry set?  //
     res++;
   return res;
}
```

Lazy Evaluation Exploitation

In a `if(a>10 && b=4)` type of thing, make sure that the first part of the AND expression is the most likely to give a false answer (or the easiest/quickest to calculate), therefore the second part will be less likely to be executed.

switch() instead of if...else...

For large decisions involving `if`...`else`...`else`..., like this:

Hide Copy Code

```
if( val == 1)
    dostuff1();
else if (val == 2)
    dostuff2();
else if (val == 3)
    dostuff3();
```

It may be faster to use a `switch`:

Hide Copy Code

```
switch( val )
{
```

```
    case 1: dostuff1(); break;

    case 2: dostuff2(); break;

    case 3: dostuff3(); break;
}
```

In the `if()` statement, if the last case is required, all the previous ones will be tested first. The `switch` lets us cut out this extra work. If you have to use a big `if`..`else`.. statement, test the most likely cases first.

Binary Breakdown

Break things down in a binary fashion, e.g. do not have a list of:

Hide Copy Code

```
if(a==1) {
} else if(a==2) {
} else if(a==3) {
} else if(a==4) {
} else if(a==5) {
} else if(a==6) {
} else if(a==7) {
} else if(a==8)

{
}
```

Have instead:

Hide Copy Code

```
if(a<=4) {
    if(a==1)    {
    }  else if(a==2)  {
    }  else if(a==3)  {
    }  else if(a==4)   {

    }
}
else
{
    if(a==5)  {
```

```
    } else if(a==6)   {
    } else if(a==7)  {
    } else if(a==8)  {
    }
}
```

Or even:

Hide Shrink Copy Code

```
if(a<=4)
{
    if(a<=2)
    {
        if(a==1)
        {
            /* a is 1 */
        }
        else
        {
            /* a must be 2 */
        }
    }
    else
    {
        if(a==3)
        {
            /* a is 3 */
        }
        else
        {
            /* a must be 4 */
        }
    }
}
else
{
    if(a<=6)
    {
        if(a==5)
        {
            /* a is 5 */
        }
        else
```

```
        {
            /* a must be 6 */
        }
    }
    else
    {
        if(a==7)
        {
            /* a is 7 */
        }
        else
        {
            /* a must be 8 */
        }
    }
}
```

Hide Copy Code

Slow and Inefficient

Hide Copy Code

```
    c=getch();
    switch(c){
        case 'A':
        {
            do something;
            break;
        }
        case 'H':
        {
            do something;
            break;
        }
        case 'Z':
        {
            do something;
            break;
        }
    }
```

Hide Copy Code

Fast and Efficient

Hide Copy Code

```
    c=getch();
    switch(c){
        case 0:
        {
            do something;
            break;
        }
        case 1:
        {
            do something;
            break;
        }
        case 2:
        {
            do something;
            break;
        }
    }
```

Compare between the two Case statements

Switch statement vs. lookup tables

The `switch` statement is typically used for one of the following reasons:

- To call to one of several functions.
- To set a variable or return a value.
- To execute one of several fragments of code.

If the `case` labels are dense, in the first two uses of `switch` statements, they could be implemented more efficiently using a lookup table. For example, two implementations of a routine that disassembles condition codes to strings:

Hide Copy Code

```
char * Condition_String1(int condition) {
  switch(condition) {
     case 0: return "EQ";
     case 1: return "NE";
     case 2: return "CS";
     case 3: return "CC";
     case 4: return "MI";
     case 5: return "PL";
     case 6: return "VS";
     case 7: return "VC";
     case 8: return "HI";
     case 9: return "LS";
     case 10: return "GE";
     case 11: return "LT";
     case 12: return "GT";
     case 13: return "LE";
     case 14: return "";
     default: return 0;
  }
}

char * Condition_String2(int condition) {
   if ((unsigned) condition >= 15) return 0;
      return
      "EQ\0NE\0CS\0CC\0MI\0PL\0VS\0VC\0HI\0LS\0GE\0LT\0GT\0LE\0\0" +
       3 * condition;
}
```

The first routine needs a total of 240 bytes, the second only 72 bytes.

Loops

Loops are a common construct in most programs; a significant amount of the execution time is often spent in loops. It is therefore worthwhile to pay attention to time-critical loops.

Loop termination

The loop termination condition can cause significant overhead if written without caution. We should always write count-down-to-zero loops and use simple termination conditions. The execution will take less time if the termination conditions are simple. Take the following two sample routines, which calculate n!. The first implementation uses an incrementing loop, the second a decrementing loop.

Hide Copy Code

```
int fact1_func (int n)
{
    int i, fact = 1;
    for (i = 1; i <= n; i++)
      fact *= i;
    return (fact);
}

int fact2_func(int n)
{
    int i, fact = 1;
    for (i = n; i != 0; i--)
       fact *= i;
    return (fact);
}
```

As a result, the second one `fact2_func`" will be more faster than the first one.

Faster for() loops

It is a simple concept but effective. Ordinarily, we used to code a simple `for()` loop like this:

Hide Copy Code

```
for( i=0;  i<10;  i++){ ... }
```

[`i` loops through the values 0,1,2,3,4,5,6,7,8,9]

If we needn't care about the order of the loop counter, we can do this instead:

Hide Copy Code

```
for( i=10; i--; ) { ... }
```

Using this code, `i` loops through the values 9,8,7,6,5,4,3,2,1,0, and the loop should be faster.

This works because it is quicker to process `i--` as the test condition, which says "Is `i` non-zero? If so, decrement it and continue". For the original code, the processor has to calculate "Subtract `i` from 10. Is the result non-zero? If so, increment `i` and continue.". In tight loops, this makes a considerable difference.

The syntax is a little strange, put is perfectly legal. The third statement in the loop is optional (an infinite loop would be written as `for( ; ; )`). The same effect could also be gained by coding:

Hide Copy Code

```
for(i=10; i; i--){}
```

or (to expand it further):

Hide Copy Code

```
for(i=10; i!=0; i--){}
```

The only things we have to be careful of are remembering that the loop stops at 0 (so if it is needed to loop from 50-80, this wouldn't work), and the loop counter goes backwards. It's easy to get caught out if your code relies on an ascending loop counter.

We can also use register allocation, which leads to more efficient code elsewhere in the function. This technique of initializing the loop counter to the number of iterations required and then decrementing down to zero, also applies to `while` and `do` statements.

Loop jamming

Never use two loops where one will suffice. But if you do a lot of work in the loop, it might not fit into your processor's instruction cache. In this case, two separate loops may actually be faster as each one can run completely in the cache. Here is an example.

Hide Copy Code

Hide Copy Code

```
//Original Code :

for(i=0; i<100; i++){
    stuff();
}

for(i=0; i<100; i++){
    morestuff();
}
```

```
//It would be better to do:

for(i=0; i<100; i++){
    stuff();
    morestuff();
}
```

Function Looping

Functions always have a certain performance overhead when they are called. Not only does the program pointer have to change, but in-use variables have to be pushed onto a stack, and new variables allocated. There is much that can be done then to the structure of a program's functions in order to improve a program's performance. Care must be taken though to maintain the readability of the program whilst keeping the size of the program manageable.

If a function is often called from within a loop, it may be possible to put that loop inside the function to cut down the overhead of calling the function repeatedly, e.g.:

Hide Copy Code

```
for(i=0 ; i<100 ; i++)
{
    func(t,i);
}
-
-
-
void func(int w,d)
{
    lots of stuff.
}
```

Could become....

Hide Copy Code

```
func(t);
-
-
```

```
-
void func(w)
{
    for(i=0 ; i<100 ; i++)
    {
        //lots of stuff.
    }
}
```

Loop unrolling

Small loops can be unrolled for higher performance, with the disadvantage of increased code size. When a loop is unrolled, a loop counter needs to be updated less often and fewer branches are executed. If the loop iterates only a few times, it can be fully unrolled, so that the loop overhead completely disappears.

This can make a **big** difference. It is well known that unrolling loops can produce considerable savings, e.g.:

Hide Copy Code

```
for(i=0; i<3; i++){
    something(i);
}

//is less efficient than
```

Hide Copy Code

```
something(0);
something(1);
something(2);
```

because the code has to check and increment the value of `i` each time round the loop. Compilers will often unroll simple loops like this, where a fixed number of iterations is involved, but something like:

Hide Copy Code

```
for(i=0;i< limit;i++) { ... }
```

is unlikely to be unrolled, as we don't know how many iterations there will be. It is, however, possible to unroll this sort of loop and take advantage of the speed savings that can be gained.

The following code (Example 1) is obviously much larger than a simple loop, but is much more efficient. The block-size of 8 was chosen just for demo purposes, as any suitable size will do - we just have to repeat the "loop-contents" the same amount. In this example, the loop-condition is tested once every 8 iterations, instead of on each one. If we know that we will be working with arrays of a certain size, you could make the block size the same

size as (or divisible into the size of) the array. But, this block size depends on the size of the machine's cache.

Hide Shrink Copy Code

```
//Example 1

#include<STDIO.H>

#define BLOCKSIZE (8)

void main(void)
{
int i = 0;
int limit = 33;  /* could be anything */
int blocklimit;

/* The limit may not be divisible by BLOCKSIZE,
 * go as near as we can first, then tidy up.
 */
blocklimit = (limit / BLOCKSIZE) * BLOCKSIZE;

/* unroll the loop in blocks of 8 */
while( i < blocklimit )
{
    printf("process(%d)\n", i);
    printf("process(%d)\n", i+1);
    printf("process(%d)\n", i+2);
    printf("process(%d)\n", i+3);
    printf("process(%d)\n", i+4);
    printf("process(%d)\n", i+5);
    printf("process(%d)\n", i+6);
    printf("process(%d)\n", i+7);

    /* update the counter */
    i += 8;

}

/*
 * There may be some left to do.
 * This could be done as a simple for() loop,
 * but a switch is faster (and more interesting)
 */
```

```
if( i < limit )
{
    /* Jump into the case at the place that will allow
     * us to finish off the appropriate number of items.
     */

    switch( limit - i )
    {
        case 7 : printf("process(%d)\n", i); i++;
        case 6 : printf("process(%d)\n", i); i++;
        case 5 : printf("process(%d)\n", i); i++;
        case 4 : printf("process(%d)\n", i); i++;
        case 3 : printf("process(%d)\n", i); i++;
        case 2 : printf("process(%d)\n", i); i++;
        case 1 : printf("process(%d)\n", i);
    }
}

}
```

Population count - counting the number of bits set

This example 1 efficiently tests a single bit by extracting the lowest bit and counting it, after which the bit is shifted out. The example 2 was first unrolled four times, after which an optimization could be applied by combining the four shifts of n into one. Unrolling frequently provides new opportunities for optimization.

Hide Shrink Copy Code

```
//Example - 1

int countbit1(uint n)
{
  int bits = 0;
  while (n != 0)
  {
    if (n & 1) bits++;
    n >>= 1;
   }
  return bits;
}


//Example - 2
```

```
int countbit2(uint n)
{
   int bits = 0;
   while (n != 0)
   {
      if (n & 1) bits++;
      if (n & 2) bits++;
      if (n & 4) bits++;
      if (n & 8) bits++;
      n >>= 4;
   }
   return bits;
}
```

Early loop breaking

It is often not necessary to process the entirety of a loop. For example, if we are searching an array for a particular item, break out of the loop as soon as we have got what we need. Example: this loop searches a list of 10000 numbers to see if there is a -99 in it.

Hide Copy Code

```
found = FALSE;
for(i=0;i<10000;i++)
{
    if( list[i] == -99 )
    {
        found = TRUE;
    }
}

if( found ) printf("Yes, there is a -99. Hooray!\n");
```

This works well, but will process the entire array, no matter where the search item occurs in it. A better way is to abort the search as soon as we've found the desired entry.

Hide Copy Code

```
found = FALSE;
for(i=0; i<10000; i++)
{
    if( list[i] == -99 )
    {
```

```
            found = TRUE;
            break;
        }
    }
    if( found ) printf("Yes, there is a -99. Hooray!\n");
```

If the item is at, say position 23, the loop will stop there and then, and skip the remaining 9977 iterations.

Function Design

It is a good idea to keep functions small and simple. This enables the compiler to perform other optimizations, such as register allocation, more efficiently.

Function call overhead

Function call overhead on the processor is small, and is often small in proportion to the work performed by the called function. There are some limitations up to which words of arguments can be passed to a function in registers. These arguments can be integer-compatible (`char`, `short`s, `int`s and `float`s all take one word), or structures of up to four words (including the 2-word `double`s and long `long`s). If the argument limitation is 4, then the fifth and subsequent words are passed on the stack. This increases the cost of storing these words in the calling function and reloading them in the called function.

In the following sample code:

Hide Copy Code

```
int f1(int a, int b, int c, int d) {
   return a + b + c + d;
}

int g1(void) {
   return f1(1, 2, 3, 4);
}


int f2(int a, int b, int c, int d, int e, int f) {
  return a + b + c + d + e + f;
}

ing g2(void) {
```

```
    return f2(1, 2, 3, 4, 5, 6);
}
```

the fifth and sixth parameters are stored on the stack in `g2`, and reloaded in `f2`, costing two memory accesses per parameter.

Minimizing parameter passing overhead

To minimize the overhead of passing parameters to functions:

- Try to ensure that small functions take four or fewer arguments. These will not use the stack for argument passing.
- If a function needs more than four arguments, try to ensure that it does a significant amount of work, so that the cost of passing the stacked arguments is outweighed.
- Pass pointers to structures instead of passing the structure itself.
- Put related arguments in a structure, and pass a pointer to the structure to functions. This will reduce the number of parameters and increase readability.
- Minimize the number of `long` parameters, as these take two argument words. This also applies to `double`s if software floating-point is enabled.
- Avoid functions with a parameter that is passed partially in a register and partially on the stack (split-argument). This is not handled efficiently by the current compilers: all register arguments are pushed on the stack.
- Avoid functions with a variable number of parameters. Those functions effectively pass all their arguments on the stack.

Leaf functions

A function which does not call any other functions is known as a leaf function. In many applications, about half of all function calls made are to leaf functions. Leaf functions are compiled very efficiently on every platform, as they often do not need to perform the usual saving and restoring of registers. The cost of pushing some registers on entry and popping them on exit is very small compared to the cost of the useful work done by a leaf function that is complicated enough to need more than four or five registers. If possible, we should try to arrange for frequently-called functions to be leaf functions. The number of times a function is called can be determined by using the profiling facility. There are several ways to ensure that a function is compiled as a leaf function:

- Avoid calling other functions: this includes any operations which are converted to calls to the C-library (such as division, or any floating-point operation when the software floating-point library is used).
- Use `__inline` for small functions which are called from it (inline functions discussed next).

Inline functions

Function inlining is disabled for all debugging options. Functions with the keyword `__inline` results in each call to an inline function being substituted by its body, instead of a normal call. This results in faster code, but it adversely affects code size, particularly if the inline function is large and used often.

Hide Copy Code

```
__inline int square(int x) {
   return x * x;
}

#include <MATH.H>

double length(int x, int y){
    return sqrt(square(x) + square(y));
}
```

There are several advantages to using inline functions:

- No function call overhead.

As the code is substituted directly, there is no overhead, like saving and restoring registers.

- Lower argument evaluation overhead.

The overhead of parameter passing is generally lower, since it is not necessary to copy variables. If some of the parameters are constants, the compiler can optimize the resulting code even further.

The big disadvantage of inline functions is that the code sizes increase if the function is used in many places. This can vary significantly depending on the size of the function, and the number of places where it is used.

It is wise to only inline a few critical functions. Note that when done wisely, inlining may decrease the size of the code: a call takes usually a few instructions, but the optimized version of the inlined code might translate to even less instructions.

Using Lookup Tables

A function can often be approximated using a lookup table, which increases performance significantly. A table lookup is usually less accurate than calculating the value properly, but for many applications, this does not matter.

Many signal processing applications (for example, modem demodulator software) make heavy use of `sin` and `cos`functions, which are computationally expensive to calculate. For real-time systems where accuracy is not very important, `sin`/`cos` lookup tables might be essential. When using lookup tables, try to combine as many adjacent operations as possible into a single lookup table. This is faster and uses less space than multiple lookup tables.

Floating-Point Arithmetic

Although floating point operations are time consuming for any kind of processors, sometimes we need to used it in case of implementing signal processing applications. However, when writing floating-point code, keep the following things in mind:

- Floating-point division is slow.

Division is typically twice as slow as addition or multiplication. Rewrite divisions by a constant into a multiplication with the inverse (For example, `x = x / 3.0` becomes `x = x * (1.0/3.0)`. The constant is calculated during compilation.).

- Use `float`s instead of `double`s.

Float variables consume less memory and fewer registers, and are more efficient because of their lower precision. Use `float`s whenever their precision is good enough.

- Avoid using transcendental functions.

Transcendental functions, like `sin`, `exp` and `log` are implemented using series of multiplications and additions (using extended precision). As a result, these operations are at least ten times slower than a normal multiply.

- Simplify floating-point expressions.

The compiler cannot apply many optimizations which are performed on integers to floating-point values. For example, `3 * (x / 3)` cannot be optimized to `x`, since floating-point operations generally lead to loss of precision. Even the order of evaluation is important: (a + b) + c is not the same as a + (b + c). Therefore, it is beneficial to perform floating-point optimizations manually if it is known they are correct.

However, it is still possible that the floating performance will not reach the required level for a particular application. In such a case, the best approach may be to change from using

floating-point to fixed point arithmetic. When the range of values needed is sufficiently small, fixed-point arithmetic is more accurate and much faster than floating-point arithmetic.

Misc tips

In general, savings can be made by trading off memory for speed. If you can cache any often used data rather than recalculating or reloading it, it will help. Examples of this would be sine/cosine tables, or tables of pseudo-random numbers (calculate 1000 once at the start, and just reuse them if you don't need truly random numbers).

- Avoid using `++` and `--` etc. within loop expressions. E.g.: `while(n--){}`, as this can sometimes be harder to optimize.
- Minimize the use of global variables.
- Declare anything within a file (external to functions) as static, unless it is intended to be global.
- Use word-size variables if you can, as the machine can work with these better (instead of `char`, `short`,`double`, bit fields etc.).
- Don't use recursion. Recursion can be very elegant and neat, but creates many more function calls which can become a large overhead.
- Avoid the `sqrt()` square root function in loops - calculating square roots is very CPU intensive.
- Single dimension arrays are faster than multi-dimension arrays.
- Compilers can often optimize a whole file - avoid splitting off closely related functions into separate files, the compiler will do better if it can see both of them together (it might be able to inline the code, for example).
- Single precision math may be faster than double precision - there is often a compiler switch for this.
- Floating point multiplication is often faster than division - use `val * 0.5` instead of `val / 2.0`.
- Addition is quicker than multiplication - use `val + val + val` instead of `val * 3`. `puts()` is quicker than `printf()`, although less flexible.
- Use `#define`d macros instead of commonly used tiny functions - sometimes the bulk of CPU usage can be tracked down to a small external function being called thousands of times in a tight loop. Replacing it with a macro to perform the same job will remove the overhead of all those function calls, and allow the compiler to be more aggressive in its optimization..
- Binary/unformatted file access is faster than formatted access, as the machine does not have to convert between human-readable ASCII and machine-readable binary. If you don't actually need to read the data in a file yourself, consider making it a binary file.
- If your library supports the `mallopt()` function (for controlling `malloc`), use it. The `MAXFAST` setting can make significant improvements to code that does a lot

of `malloc` work. If a particular structure is created/destroyed many times a second, try setting the `mallopt` options to work best with that size.

Last, but definitely not least - turn compiler optimization on! Seems obvious, but is often forgotten in that last minute rush to get the product out on time. The compiler will be able to optimize at a much lower level than can be done in the source code, and perform optimizations specific to the target processor.

References

- Writing Efficient C for ARM
  - Document number: ARM DAI 0034A
  - Issued: January 1998
  - Copyright Advanced RISC Machines Ltd. (ARM) 1998
- Richard's C Optimization page OR: How to make your C, C++ or Java program run faster with little effort.
- Code Optimization Using the GNU C Compiler By Rahul U Joshi.
- Compile C Faster on Linux [Christopher W. Fraser (Microsoft Research), David R. Hanson (Princeton University)]
- CODE OPTIMIZATION - COMPILER [1] [2]

[Thanks to Craig Burley for the excellent comments. Thanks to Timothy Prince for the note on architectures with Instruction Level Parallelism].

- An Evolutionary Analysis of GNU C Optimizations [Using Natural Selection to Investigate Software Complexities by Scott Robert Ladd. Updated: 16 December 2003]

Other URLs

- http://www.xs4all.nl/~ekonijn/loopy.html
- http://www.public.asu.edu/~sshetty/Optimizing_Code_Manual.doc
- http://www.abarnett.demon.co.uk/tutorial.html

License

About the Author

Koushik Ghosh

Software Developer (Senior) ibm
India 🇮🇳