

Guidelines for improving CPP performance

Izaq Yosi

February 8, 2006

Contents

1 General	2
1.1 compiler optimization	2
1.2 Profilers	3
1.3 Inlining	3
2 Temporary objects	4
2.1 The Why and When	4
2.2 The operator $\xi=$	4
3 IO	5
3.1 Flush	5
4 String comparisons	5
5 Loops	6
5.1 Loop unrolling	6
5.2 Avoid redundant calculation	7
6 RVO	7
7 Type resolution	9
7.1 Virtual functions	9
7.2 Templates	9
8 CTORs and DTORs	9
8.1 inheritance	9
8.2 Construct on demand	9
8.2.1 Example	10
8.3 Initialization list	10

9 STL	10
9.1 Containers	11
9.1.1 Array	11
9.1.2 Vector	11
9.1.3 List	11
9.1.4 Dequeue	11
10 Advanced and rare points	12
10.1 Dynamic programming	12
10.2 Arithmetic	12
10.3 Pre-Increment	13
10.4 Memory management	13
11 Acknowledgments	13

1 General

1.1 compiler optimization

Most compilers have various options for optimization. Both for exe speed and size. For GCC there are the -O flags, -O1, -O2, -O3 are the levels of optimization and -Os for reducing the executable size. Common optimizations are inlining and CSE (common subexpression elimination), loop unrolling and scheduling(rearrange instructions for pipelining and parallelism). Here's the breakdown for the speed flags.

- -O1 This level turns on the most common forms of optimization that do not require any speed-space tradeoffs. The more expensive optimizations, such as instruction scheduling, are not used at this level. Compiling with the option -O1 can often take less time than compiling w/o optimization, due to the reduced amounts of data that need to be processed after simple optimizations.
- -O2 This option turns on further optimizations, in addition to those used by -O1. These additional optimizations include instruction scheduling. Only optimizations that do not require any speed-space tradeoffs are used, so the executable should not increase in size. The compiler will take longer to compile programs and require more memory than with -O1. This option is generally the best choice for deployment of a program, because it provides maximum optimization without increasing the executable size. It is the default optimization level for releases of GNU packages.
- -O3 This option turns on more expensive optimizations, such as function inlining, in addition to all the optimizations of the lower levels -O2 and -O1. The -O3 optimization level may increase the speed of the resulting executable, but can also increase its size. Under some circumstances where

these optimizations are not favorable, this option might actually make a program slower.

- -funroll-loops This option turns on loop-unrolling, and is independent of the other optimization options. It will increase the size of an executable. Whether or not this option produces a beneficial result has to be examined on a case-by-case basis.

1.2 Profilers

Need I say more?

Well, make sure you use them.

They will help find out which functions are using most of the time, in other words - the speed bottlenecks. Allowing you to optimize them.

1.3 Inlining

Function calls are expensive. They require saving various registers (SP, LR, IP, FP, AP and sometimes more) upon invocation and restoring them upon return. They also inhibit pipelining optimization because they contain branching for the called function IP (and for the calling method IP upon return). Inlining allows the compiler to plant the function code explicitly, w/o branching, in the calling function, thus eliminating register save and restore cycles (up to 100 cycles) and enabling pipelining.

To inline a method either put it's definition together with it's declaration or use the *inline* modifier. As in.

```
class hughNum {  
// Inline getter.  
int getValue(){  
return _value;  
}  
}  
or  
inline int hughNum::getValue(){  
return _value;  
}
```

Caution must be taken to inline only small methods with minimum branching (a condition or two at most). The reason is that inlined functions with many instructions will inflate the code. Bigger code might have more page faults and cache failures. So it might have less instructions in all but it will take longer.

2 Temporary objects

2.1 The Why and When

They are very common. They are not visible in code, because the compiler generated them behind the scenes. So you better be aware of when they are generated to avoid redundant construction and destruction of temporaries.

I will list a few common situations that might generate temporary objects.

- Pass by value The compiler will generate a temporary copy of the parameter. The method will work on it. When it ends it will be destroyed. If possible, prefer pass by reference to pass by value
- Return by value Similar to the previous item. The compiler will generate a temporary object to hold the returned object. It is used for left hand assignment. If possible prefer return by reference to return by value.
- Type mismatch When the compiler expects object of some type but instead has another object type it will generate code that generates a temporary object. For example:

```
hughNum myNum = 150000;
```

Assuming that hughNum defines a constructor for integer but doesn't define an assignment operator for int the compiler will generate code for creating a temporary hughNum and then assign it.

There are two ways to prevent the compiler from doing this. The first is use *explicit* directive, which instructs the compiler not to do implicit conversion. Example:

```
explicit hughNum (int number)
{
    //Initialization code.
}
```

The second is to add an assignment operator that accepts the other type.

2.2 The operator $\xi=$

Use the = operator for addition, multiplication etc. to avoid the creation of temporaries. Consider the following code for adding *hughNum* objects:
hn1 = hn2 + hn3 + hn4;

Here the compiler generated two temporary objects. The first for *temp1 = hn3+hn4* and the second for *temp2 = hn2+ temp1*; then *hn1 = temp2*; They can be eliminated by rewriting the code as:

```
hn1 = hn2  
hn1+= hn3  
hn1+= hn4;
```

For this you need to add the operator`+=` for `hughNum`.

3 IO

There are some common pitfalls to beware of.

3.1 Flush

Careful implicit buffer flush operations. for example “`endl`” stands for both carriage return and flush (system call to `write()`) which is expensive. Consider the following examples for output:

1. `cout << some_string << endl;` Entails an implicit flush operation.
2. `cout << some_string << "\n";` Output a one character string of carriage return. Can be up to ten times faster then the first example!
3. `cout << some_string << '\n';` Output one character for carriage return. The most efficient of the three.

4 String comparisons

In cases when the application has lot’s of string comparison operations related to string constructions the following recommendation can improve performance.

Instead of comparing right away compare only the prefix, say first letter, if it’s the same then do the full comparison this can save up to $1 - 1/26$ redundant comparisons (more accurately, it depends on the letter frequency in the string domain). Taking it one step further, it’s possible to calculate checksums for the strings and use them for comparison.

Example Don’t do:

```
strcmp(first_string, second_string);
```

rather do:

```
*first_string != *second_string ? *first_string - *second_string : strcmp(first_string,  
second_string);
```

or with checksums, using Adler 32 for fast CRC:

Calculate checksum once!

```
CAdler32::Calculate( first_cksum, (const unsigned char*) first_string, strlen(  
string) );
```

...

Then compare as:

```
first_cksum != second_cksum ? first_cksum - second_cksum : strcmp(first_string,  
second_string);
```

5 Loops

5.1 Loop unrolling

This technique improves speed but increases executable size. In loops the loop condition is checked for each iteration and because of the jump command it's not possible for CPU to perform pipelining. By performing the loop unrolling the compiler eliminates the condition. A simple Example:

```
for (i = 0; i < 8; i++)  
{  
    y[i] = i;  
}
```

And the unrolled version.

```
y[0] = 0;  
y[1] = 1;  
y[2] = 2;  
y[3] = 3;  
y[4] = 4;  
y[5] = 5;  
y[6] = 6;  
y[7] = 7;
```

A somewhat more general example, for unknown upper bound.

It's possible to unroll part of the iterations. Care must be taken to handle the start and end conditions correctly. The following loop.

```
for (i = 0; i < n; i++)  
{  
    y[i] = i;  
}
```

can be rewritten as:

```
//Handle start condition. Namely when n is odd handle i = 0 case.  
for (i = 0; i < (n % 2); i++)  
{  
    y[i] = i;  
}  
//Handle rest of conditions.  
for ( ; i + 1 < n; i += 2) /* no initializer */  
{  
    y[i] = i;
```

```

y[i+1] = i+1;
}

```

This cuts by half the condition checks and allows parallelism of two instructions in a row.

5.2 Avoid redundant calculation

Make sure that there are no redundant calculations inside the body of a loop. Make sure that there aren't redundant memory allocations in loop. Make sure that there are no redundant temporary objects created in the loop. Example of bad loop.

```

while( condition) {
    hughNum temp = 200000+14582211+1322492; // do some calculation on temp.
    ObjectFactory of;
    of.batch_generate(temp);
}

```

Also avoid pointer dereferencing inside loop body. For example:

```

for (int i=0; i<hughNum; ++i) {
    warehaouse->compartment->shelf->contents[i] = someValue;
}

```

should be written as:

```

Contents * contents_pointer = warehaouse->wing->compartment->shelf->contents;
for (int i=0; i<hughNum; ++i){
    contents_pointer[i] = someValue;
}

```

6 RVO

This is the return value optimization. For functions that return object by value the compiler will rewrite them as void functions with a (hidden) `_result` parameter passed by reference. The optimization uses this parameter instead of an object create locally, thus saving it's construction and destruction time.

An example is in place.

suppose that we have a member function:

```

hughNum hugeNum::operator/ (const hughNum& first, const hughNum& second);

```

implemented as:

```

hughNum operator/ (const hughNum& first, const hughNum& second)
{
    hughNum retVal;
    //Do the division using large number division algorithm and store the result in

```

```

retVal;
...
return retVal;
}

```

When use, as in: `third = first/second;`, the compiler will create the said `_result` `hughNum` object, pass it by reference and use `hughNum` assignment operator to assign it to `third`.

The compiler generates code along the lines of:

```

struct hughNum _result;
hughNum_div(_result, first, second); // pass by ref.
third = _result; // assign result to third

```

And the division operator code:

```

void hughNum_div(const hughNum& _result, const hughNum& first, const
hughNum& second)
{
    struct hughNum retVal;
    retVal.hughNum::Complex();
    //Calculate actual division into retVal
    ...
    // Copy calculation result to _result
    _result.hughNum::hughNum(retVal); // Copy constructor
    retVal.hughNum::hughNum(); // clean retVal
    return;
}

```

The optimization is achieved by eliminating the local object `retVal`, using the temporary `_result` instead. The optimized compiler generated code is:

```

void hughNum_div(const hughNum& _result, const hughNum& first, const
hughNum& second)
{
    //Calculate actual division into _result!
    ...
    return;
}

```

Instructing the compiler to perform this optimization doesn't guarantee it will happen. Experience shows that the following form is most likely to ensure this optimization will happen.

```

void hughNum_div(const hughNum& _result, const hughNum& first, const
hughNum& second)
{
    ...
}

```

```
return hughNum/*Calculate the division data and feed it to hughNum CTOR
*/;
}
```

7 Type resolution

7.1 Virtual functions

Very useful for relieving the programmer of explicit dynamic type resolution. There are three factors by which they impact performance. First is the vptr (pointer to the virtual function table), adding cost for initializing it at construction and removing it in destructor. Second, is that virtual functions can't be called by address (offset in class). They are called by pointer indirection, first the pointer to the virtual function table is fetched, then the function is accessed at its offset. The last factor is the most important, it prevents inlining. Inlining is done in compile time. Resolution of virtual functions is done at run time. Hence, the compiler cannot inline virtual functions.

The first two factors are less significant because to achieve dynamic type resolution this price will be paid anyway. For example, if we put a type member variable in the class for determining the type at run time, we then need to initialize it at the constructor. This is equivalent to initializing the vptr. Then at run time we need to perform a switch case for selecting the correct type, which is equivalent to retrieving the function address from the function table.

7.2 Templates

Inheritance allows dynamic binding. We know that it has its cost, because it prevents inlining. On the other hand using templates means that type resolution is done at compile time, which allows inlining. Hence, when possible it is advised to use templates instead of inheritance.

8 CTORs and DTORs

8.1 inheritance

Inheritance adds cycles for construction and destruction since base classes are created/destroyed. It's important to keep that in mind and don't impose inheritance relation where they're overkill.

8.2 Construct on demand

Delay object creation until the point of first usage. It is very common that objects are created but not always used.

8.2.1 Example

```
Lunch big_lunch;  
...  
if (employee.is_hungry()) { employee.eat(big_lunch); }  
else { employee.work(); }
```

Assuming employees are not always hungry it's wasteful to create the lunch anyway.

8.3 Initialization list

use them for initializing class types. Thus eliminating call to default constructor and assignment operator.

Consider the following example:

```
Class Employee {  
public:  
Employee( const char * s) name = s;  
...  
private:  
string name;  
};
```

When Employee constructor is called, before its body is executed it initializes name using the default constructor. Then in the body it activates the assignment operator. Using initialization list will eliminate the call to default constructor. As in:

```
Class Employee {  
public:  
Employee( const char * s) : name(s)  
...  
private:  
string name;  
};
```

9 STL

STL containers and algorithms are guaranteed to perform in given complexity (usually asymptotic as in big O notation). The main thing about using STL for good performance is to match the most fitting containers and algorithms to the nature of the algorithms. Following I will give a brief description of STL containers. For helping determine which container fits a given task. For further details please refer to [?, 7]

9.1 Containers

9.1.1 Array

Arrays are consecutive container of a pre determined size. Arrays permit efficient, $O(1)$, random access but not efficient insertion and deletion of elements (which are $O(n)$). Linked lists have the opposite trade-off. Consequently, Arrays are most appropriate for storing a fixed amount of data which will be accessed in an unpredictable fashion, and linked lists are best for a list of data which will be accessed sequentially and updated often with insertions or deletions. Iterating through an Array has good locality of reference, and so is much faster than iterating through a linked list of the same size, which tends to jump around in memory. When Array is accessed randomly this advantage is not guaranteed though. Array is compact. It has very little overhead for storing objects. An array of n objects size is $n * \text{memsize}(\text{object}) + 4$. The added four bytes are for the array pointer. For comparison, pointer based containers (such as List), have the pointers overhead. This overhead is more significant for small objects (chars, ints etc) and less significant for large objects.

9.1.2 Vector

The vector is also a sequential container like the array. It is allocated to an initial size but it can expand beyond this size when needed. It supports random access to elements. Insertion and removal of elements from the end are at complexity $O(1)$. Deletion and other types of insertion are at complexity $O(n)$. For insertion though a possible additional action is to increase the vector size this can be an expensive operation. It doesn't affect the asymptotic complexity but it has a practical impact.

9.1.3 List

List is a doubly linked list. That is, it is a Sequence that supports both forward and backward traversal, and (amortized) constant time insertion and removal of elements at the beginning or the end, or in the middle. Singly linked lists, which only support forward traversal, are also sometimes useful. If you do not need backward traversal, then slist may be more efficient than list.

9.1.4 Dequeue

Is very similar to vector, like vector, it is a sequence that supports random access to elements, constant time ($O(1)$), insertion and removal of elements at the end of the sequence, and linear time, ($O(n)$), insertion and removal of elements in the middle.

10 Advanced and rare points

10.1 Dynamic programming

This is a general optimization technique. It is not limited to C++. It applies to computation problems that are made of overlapping sub problems and have optimal substructure (optimal solution of sub problems can be used to calculate the optimal solution). It usually either takes either a top down approach or bottom up approach. In top down the problem is broken into sub problems (which are broken again and again until solved), their solution is memorized and used to calculate the solution. In bottom up all the relevant sub problems are first calculated and then used to solve the larger problems and last the problem.

This is just the tip of the iceberg for a very efficient optimization technique. For further details please refer to the programmers “bible”[6].

10.2 Arithmetic

Simplify (for the compiler that is) expressions.

Examples:

1. $x * y + x * z = x * (y + z)$; One multiplication less.
2. $y/x + z/x = (1/x) * (y + z)$; Instead of two divisions we have one division and a multiplication. Since divisions are slower we improve speed.
3. $y/x + z/x = (y + z)/x$; and now we have only one division ...

same goes for logical expressions. $((x||y)&&z) = (z&&(x||y))$; You know that C++ uses lazy evaluation. It's guaranteed by the standard so this works. This can be extended to any compound condition, put the simple conditions on the right hand side for they will be evaluated first. Be careful though not to break logic that assume lazy evaluation, like if the first condition check for pointer validity don't move it!

Calculate the integer square root for integers. To find the integer square root of an integer subtract successive odd numbers until the result is $\neq 0$. The number of successful subtractions is the integer square root of the integer. This method certainly is faster than the usual method of finding square root but can be used only at places where “Integer only” Square Root of “Integers” is required. For example:

9

$9 - 1 = 8$

$8 - 3 = 5$

$5 - 5 = 0$

3 - number of subtractions, is the integer square root of 9 .

Multiplication and Division by a Power of 2. Use bit shift which is faster for multiplication and more so for division. Note though that it only works for integers.

This can be done because of the following equalities.

$$x \ll y = x * 2^y$$

$$x >> y = x / 2^y$$

10.3 Pre-Increment

Pre increment and decrement operators are more efficient than their Post counter parts. Postfix operators first copy the value to a temporary object, then increment the value and return the temporary. That's because their behavior is "use value first, increment for later". On the other hand prefix operators first increment the value and then return a reference to it. For primitive types (ints for example) this difference is negligible. However for iterators it might be noticeable.

10.4 Memory management

Default `new()` and `delete()` are multipurpose memory management methods. They can handle allocation of fixed size objects and variable size objects. They ensure thread safety in multithreaded environment. This means that if your code is made up of many memory allocations and deallocations and it doesn't need a general purpose memory management it's possible to considerably improve performance by writing a specific memory management scheme. for more details please refer to chapters six and seven of [2].

11 Acknowledgments

This document was built during the work for capacity improvement of SFE applications for Comverse©, SMSC product. It was presented to the developers as guidelines for both refactoring and developing of new code. Most items presented at this document were implemented during the capacity improvement process. The content of this document is heavily influenced by real life example from the source code and from quantifying different bits and pieces of it.

This is of course an important subject covered by many articles and books. The lore they impart is priceless, for further reading refer to the bibliography.

References

- [1] Effective C++: 50 Specific Ways to Improve Your Programs and Design, Scott Meyers.
- [2] Efficient C++ Performance Programming Techniques, Dov Bulka and David Mayhew.
- [3] GCC 4.0 online manual.
- [4] How To Optimize C/C++ Source - Performance Programming, Sachin Garg.
- [5] IBM[®], rational quantify manual.
- [6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0262032937. Chapter 15: Dynamic Programming, pp.323369.
- [7] Alexander Stepanov and Meng Lee. The Standard Template Library (STL). ANSI/ISO, October 1994.