Optimizing C and C++ Code

Embedded software often runs on processors with limited computation power, thus optimizing the code becomes a necessity. In this article we will explore the following optimization techniques for C and C++ code developed for Real-time and Embedded Systems.

10. Don't define a return value if not used
11. Consider locality of reference for code and data
12. Locality of reference in multi-dimensional arrays
13. Prefer int over char and short
14. Define lightweight constructors
15. Prefer initialization over assignment
16. Use constructor initialization lists
17. Do not declare "just in case" virtual functions
18. In-line 1 to 3 line functions
19. Avoid cascaded function calls
20. Prefer preincrement over postincrement
21. Define move constructors in C++11
22. Use hardware accelerators and SIMD hardware
23. Use profile guided optimization

Many techniques discussed here have roots in the material we covered in the articles dealing with C to Assembly translation. A good understanding of the following articles will help:

- C To Assembly Translation
- C To Assembly Translation II
- C To Assembly Translation III

Premature optimization is the root of all evil

Donald Knuth wrote, "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

In general, correctness and readability considerations trump code performance issues for most of your code. For a small fraction of your code, you may have to sacrifice readability to improve performance. Such optimizations should be carried out when the project is nearing completion. You have a better idea of the performance critical code when you have a working system.

That said, it is important to recognize that many optimization techniques are just sound programming practices as they improve performance as well as code readability. Such techniques should be applied right from the project start.

Adjust structure sizes to power of two

When arrays of structures are involved, the compiler performs a multiply by the structure size to perform the array indexing. If the structure size is a power of 2, an expensive multiply operation will be replaced by an inexpensive shift operation. Thus keeping structure sizes aligned to a power of 2 will improve performance in array indexing.

Place case labels in narrow range

If the case labels are in a narrow range, the compiler does not generate an if-else-if cascade for the switch statement. Instead, it generates a jump table of case labels along with manipulating the value of the switch to index the table. This code generated is faster than if-else-if cascade code that is generated in cases where the case labels are far apart. Also, performance of a jump table based switch statement is independent of the number of case entries in switch statement.

Place frequent case labels first

If the case labels are placed far apart, the compiler will generate if-else-if cascaded code with comparing for each case label and jumping to the action for leg on hitting a label match. By placing the frequent case labels first, you can reduce the number of comparisons that will be performed for frequently occurring scenarios. Typically this means that cases corresponding to the success of an operation should be placed before cases of failure handling.

Break big switch statements into nested switches

The previous technique does not work for some compilers as they do not generate the cascade of if-else-if in the order specified in the switch statement. In such cases nested switch statements can be used to get the same effect.

To reduce the number of comparisons being performed, judiciously break big switch statements into nested switches. Put frequently occurring case labels into one switch and keep the rest of case labels into another switch which is the default leg of the first switch.

Splitting a switch statement

```
1. // This switch statement performs a switch on frequent messages and handles the
2. // infrequent messages with another switch statement in the default leg of the outer
3. // switch statement
4.
5. pMsg = ReceiveMessage();
```

```
 6.  switch (pMsg->type)
 7.  {
 8.  case FREQUENT_MSG1:
 9.     handleFrequentMsg1();
10.     break;
11.
12.  case FREQUENT_MSG2:
13.     handleFrequentMsg2();
14.     break;
15.
16.  . . .
17.
18.  case FREQUENT_MSGn:
19.     handleFrequentMsgn();
20.     break;
21.
22.  default:
23.     // Nested switch statement for handling infrequent messages.
24.     switch (pMsg->type)
25.     {
26.     case INFREQUENT_MSG1:
27.        handleInfrequentMsg1();
28.        break;
29.
30.     case INFREQUENT_MSG2:
31.        handleInfrequentMsg2();
32.        break;
33.
34.       . . .
35.
36.     case INFREQUENT_MSGm:
37.        handleInfrequentMsgm();
38.        break;
39.     }
40.  }
```

Minimize local variables

If the number of local variables in a function is less, the compiler will be able to fit them into registers. Hence, it will be avoiding frame pointer operations on local variables that are kept on stack. This can result in considerable improvement due to two reasons:

- All local variables are in registers so this improves performance over accessing them from memory.
- If no local variables need to be saved on the stack, the compiler will not incur the overhead of setting up and restoring the frame pointer.

Declare local variables in the inner most scope

Do not declare all the local variables in the outermost function scope. You will get better performance if local variables are declared in the inner most scope. Consider the example below; here object "a" is needed only in the error case, so it should be invoked only inside the error check. If this parameter was declared in the outermost scope, all function calls would have incurred the overhead of object a's creation (i.e. invoking the default constructor for "a").

Local variable scope

```
 1. int foo(char *pName)
 2. {
 3.     if (pName == NULL)
 4.     {
 5.        A a;
 6.        ...
 7.        return ERROR;
 8.      }
 9.      ...
10.      return SUCCESS;
11. }
```

Reduce the number of parameters

Function calls with large number of parameters may be expensive due to large number of parameter pushes on stack on each call. For the same reason, avoid passing complete structures as parameters. Use pointers and references in such cases.

Use references for parameter passing and return value for types bigger than 4 bytes

When parameters are passed by value, the complete parameter memory is copied on the stack. This is fine for regular types like integer, pointer etc. These types are generally restricted to four bytes. When passing bigger types, the cost of copying the object on the stack can be prohibitive. In case of classes there will be an additional overhead of invoking the constructor for the temporary copy that is created on the stack. When the function exits the destructor will also be invoked.

Thus it is efficient to pass references as parameters. This way you save on the overhead of a temporary object creation, copying and destruction. This optimization can be performed easily without a major impact to the code by replacing pass by value parameters by const references. (It is important to pass const references so that a bug in the called function does not change the actual value of the parameter.

Passing bigger objects as return values also has the same performance issues. A temporary return object is created in this case too.

Don't define a return value if not used

The called function does not "know" if the return value is being used. So, it will always pass the return value. This return value passing may be avoided by not defining a return value which is not being used.

Consider locality of reference for code and data

The processor keeps data or code that is referenced in cache so that on its next reference if gets it from cache. These cache references are faster. Hence it is recommended that code and data that are being used together should actually be placed together physically. This is actually enforced into the language in C++. In C++, object's data is stored in contigueous memory, thus improving locality of reference for data. This also applies to code, as most methods that deal with the object will be stored in contiguous memory.

Locality of reference can be improved for C code as well. The declaration order of related code and functions can be arranged so that closely coupled code and data are declared together.

Locality of reference in multi-dimensional arrays

When working with two dimensional arrays, organize the algorithm so that your inner most loop iterates over the second index.

```
1.  int a[MAX_X][MAX_Y];
```

The above array will be organized in the memory as:

```
1.  a[0][0] a[0][1] ... a[0][MAX_Y-1] a[1][0] a[1][1] ... a[1][MAX_Y-1].....a[MAX_X-1][0] a[MAX_X-1][1] ... a[0][MAX_Y-1]
```

If you have accessed `a[x][y]`, accessing `a[x][y+1]` will be faster than accessing `a[x+1][y]`. The entry `a[x][y+1]` will probably be read from the processor cache. `a[x+1][y]` would most likely need to fetched from the external memory. Keep in mind that sequential reading of memory is always

faster than random access as the processor would have probably fetched a complete cache line.

Prefer int over char and short

With C and C++ prefer use of int over char and short. The main reason behind this is that C and C++ perform arithmetic operations and parameter passing at integer level. If you have a value that can fit in a byte, you should still consider using an int to hold the number. If you use a char, the compiler will first convert the values into integer, perform the operations and then convert back the result to char.

Lets consider the following code which presents two functions that perform the same operation with char and int.

Comparing char and int operations

```
 1. char sum_char(char a, char b)
 2. {
 3.     char c;
 4.     c = a + b;
 5.     return c;
 6. }
 7.
 8. int sum_int(int a, int b)
 9. {
10.     int c;
11.     c = a + b;
12.     return c;
13. }
```

A call to sum_char involves the following operations:

1. Convert the second parameter into an int by sign extension (C and C++ push parameters in reverse)
2. Push the sign extended parameter on the stack as b.
3. Convert the first parameter into an int by sign extension.
4. Push the sign extended parameter on to the stack as a.
5. The called function adds a and b
6. The result is cast to a char.
7. The result is stored in char c.
8. c is again sign extended
9. Sign extended c is copied into the return value register and function returns to caller.

10. The caller now converts again from int to char.
11. The result is stored.

A call to sum_int involves the following operations:

1. Push int b on stack
2. Push int a on stack
3. Called function adds a and b
4. Result is stored in int c
5. c is copied into the return value register and function returns to caller.
6. The called function stores the returned value.

Thus we can conclude that int should be used for all integer variables unless storage requirements force us to use a char or short. When char and short have to be used, consider the impact of byte alignment and ordering to see if you would really save space. (Many processors align structure elements at 16 byte boundaries)

Define lightweight constructors

Keep the constructor light weight. The constructor will be invoked for every object creation. Keep in mind that many times the compiler might be creating temporary object over and above the explicit object creations in your program. Thus optimizing the constructor might give you a big boost in performance. If you have an array of objects, the default constructor for the object should be optimized first as the constructor gets invoked for every object in the array.

Prefer initialization over assignment

Consider the following example of a complex number::

Initialization and assignment

```
1. void foo()
2. {
3.     Complex c;
4.     c = (Complex)5;
5. }
6.
7. void foo_optimized()
8. {
```

```
 9.      Complex c = 5;
10.  }
```

In the function foo, the complex number c is being initialized first by the instantiation and then by the assignment. In foo_optimized, c is being initialized directly to the final value, thus saving a call to the default constructor of Complex.

Use constructor initialization lists

Use constructor initialization lists to initialize the embedded variables to the final initialization values. Assignments within the constructor body will result in lower performance as the default constructor for the embedded objects would have been invoked anyway. Using constructor initialization lists will directly result in invoking the right constructor, thus saving the overhead of default constructor invocation.

In the example given below, the optimized version of the Employee constructor saves the default constructor calls for m_name and m_designation strings.

Constructor initialization lists

```
 1.  Employee::Employee(String name, String designation)
 2.  {
 3.      m_name = name;
 4.      m_designation = designation;
 5.  }
 6.
 7.  /* === Optimized Version === */
 8.
 9.  Employee::Employee(String name, String designation): m_name(name), m_destignation (designation)
10.  {
11.  }
```

Do not declare "just in case" virtual functions

Virtual function calls are more expensive than regular function calls so do not make functions virtual "just in case" somebody needs to override the default behavior. If the need arises, the developer can just as well edit the additional base class header file to change the declaration to virtual.

In-line 1 to 3 line functions

Converting small functions (1 to 3 lines) into in-line will give you big improvements in throughput. In-lining will remove the overhead of a function call and associated parameter passing. But using this technique for bigger functions can have negative impact on performance due to the associated code bloat. Also keep in mind that making a method inline should not increase the dependencies by requiring a explicit header file inclusion when you could have managed by just using a forward reference in the non-inline version. (See the article on header file include patterns for more details).

Avoid cascaded function calls

A disturbingly common practice in coding is to cascade function calls that return pointers or references. Refer to the following code:

Cascaded function calls

```
1.  A::RaiseAlarmForFailedTerminals()
2.  {
3.     for (i=0; i <MAX_TERMINALS; i++)
4.     {
5.        if (GetX().GetY().GetZ().status == OUT_OF_SERVICE)
6.        {
7.             RaiseAlarm(GetX().GetY().GetZ().status,
8.                        GetX().GetY().GetZ().cause);
9.        }
10.    }
11. }
```

The code above has poor readability and the compiler cannot optimize the common sub-expression as the compiler cannot assume that `GetX()`, `GetY()` and `GetZ()` functions will return the same reference every time.

The following code optimizes the code by storing the reference to `Z` and using it within the loop.

Optimized by storing the cascade in a variable

```
1.  A::RaiseAlarmForFailedTerminals()
2.  {
3.     Z &z = GetX().GetY().GetZ();
4.     for (i=0; i <MAX_TERMINALS; i++)
5.     {
6.        if (z.status == OUT_OF_SERVICE)
7.        {
8.             RaiseAlarm(z.status, z.cause);
```

```
 9.        }
10.    }
11. }
```

Prefer preincrement over postincrement

For classes that overload the pre-increment and postincrement operators, using the pre-increment operators is more efficient. An overloaded preincrement operator's code would be:

1. Create a copy of the object,
2. increment the variable and
3. Return the variable created in step 1.

A preincrement operator avoids creating a temporary. An operator's code would be:

1. Increment the variable
2. Return the variable

Define move constructors in C++11

A C++ 03 compiler always calls the copy constructor when it needs to make a copy of an object. This can be wasteful in scenarios where the object being copied is a temporary. Consider the case of a `remote_integer` class that stores a reference to an object saved on the heap (refer to the following code block). One of the constructors of this class takes an integer and allocates memory to store this integer. Now consider what happens when function `foo()` returns `remote_integer` to `bar()`; the compiler creates a temporary copy of foo_ri by calling the copy constructor. This temporary object gets copied once more in the bar function when bar_ri stores the return value.

A function returning `remote_integer`

```
1. remote_integer(const int n)
2. {
3.     m_p = new int(n);
4. }
5.
6. remote_integer foo(const remote_integer& other
7. {
8.     remote_integer foo_ri=5;
9.     ...
```

```
10.     return foo_ri;
11. }
12.
13. void bar()
14. {
15.     remote_integer bar_ri = foo();
16. }
```

The redundant copies happen because the compiler has no way of telling the user that a copy is being made from a temporary object that does not need to be preserved.

A copy constructor

```
1.  remote_integer(const remote_integer& other)
2.  {
3.      if (other.m_p)
4.      {
5.         m_p = new int(*other.m_p);
6.      }
7.      else
8.      {
9.         m_p = nullptr;
10.     }
11. }
```

C++ 11 adds the concept of a move constructor. The compiler will call a move constructor (if one is defined) when it is copying from a temporary object. This gives the user an option to forgo the copying and literally highjack the contents of the temporary object. A move constructor is shown below. Note the absence of a const and a double ampersand in the constructor signature. In our example, the move constructor for the `remote_integer` does not allocate any new memory; it just copies the pointer from the passed `remote_integer` special reference (called Rvalue reference).

A move constructor

```
1.  remote_integer(remote_integer&& other)
2.  {
3.      m_p = other.m_p;
4.      other.m_p = nullptr;
5.  }
```

Revisit the `foo()` and `bar` example. In a function return, the compiler will call the move constructor when returning a temporary in `foo()` and saving the return value in `bar`. Since the move constructor was used, the memory allocated in the `remote_integer foo_ri=5;` line of `foo()` would be still in use. Thus we saved two memory allocations in just returning a value. The benefits of this will accrue in passing parameters and internal STL copies.

C++ 11 implementation of the standard library has been optimized to take full advantage of move constructors. Implementing move constructors in classes that are used in STL containers will improve performance. For details refer to the video on Rvalue references.

Use hardware accelerators and SIMD hardware

Perform some spring cleaning on your compiler options. Check if new hardware and performance and parallelization options are enabled. For example, Intel processors support AVX, 256-bit SIMD unit. Intel/AMD processors also support SSE 4.2.

If your platform is equipped with a GPU, performance of algorithms that blend themselves to parallelization can be improved with CUDA (NVIDIA) or C++ AMP (Microsoft). C++ AMP can also be used to improve performance when no GPU is available.

Use profile guided optimization

Compilers from Microsoft and Intel support profile guided optimization (POGO). With POGO, compilers use the information from actual program execution to find the areas that need to be optimized for speed. The compiler can also use the loop execution and branching history from previous runs to generate the optimal code for loops and branching.

Related Links

- C To Assembly Translation
- C To Assembly Translation II
- C To Assembly Translation III
- Byte Alignment and Ordering
- Header File Include Patterns
- C++ to C Mapping

EventStudio

- call flow gallery
- sequence diagrams
- use cases & more

- testimonials
- download free trial

VisualEther

- Wireshark gallery
- visualize Wireshark
- auto diagnose
- select fields
- download free trial

Telecom+networking

- LTE tutorials and call flows
- IMS call flows
- telecom call flows
- TCP/IP protocol flows

Software Design

- object oriented design
- design patterns
- embedded design
- fault handling
- congestion control

Follow

- facebook
- twitter
- linkedin
- tumblr
- google+

Share

Company

- contact us
- blog

© 2015 EventHelix.com Inc.