Code optimization techniques for better CPU utilization

As we all know that CPU (Central Processing Unit) act as brain of the computer system as it carries out the instructions of a computer program to perform the basic arithmetical, logical, and input/output operations of the system.

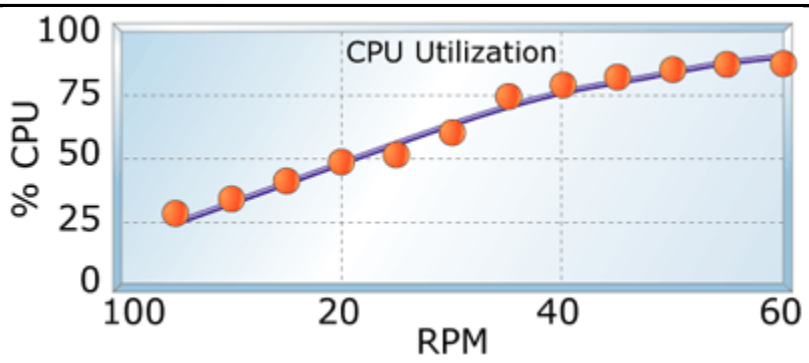
In today's competitive embedded world one of the most important factors while designing any system would be customizing the software for a better CPU utilization.

CPU utilization: As an example if a hard disk is transferring data over the interface to the rest of the system, it uses some of the system's resources. One of these critical resources is how much CPU time is required for a particular transfer. Because higher the percentage of the CPU used by the data transfer the less execution time CPU can devote to other tasks. This time utilization is generally known as CPU utilization. The CPU time is often measured in clock ticks or as a percentage of the CPU's capacity. It is used to measure the CPU workload of a program.

CPU utilization will provide the details regarding how much CPU is busy at that moment. For example,

a)100%busy
b) 50% busy

If the system says that 50% utilized that means a process wanting to be on the CPU will have a 50/50 chance of getting scheduled right away, process has a 1 in 2 chance of getting on the CPU, a 1 in 2 chance of being made to wait for the CPU. So in other words CPU utilization means keeping CPU free as much as possible.



The current embedded field is owned by the following RTOs because of their own special qualities,

1. WinCE
2. RTLinux
3. VxWorks
4. ThreadX etc

The major feature of RTOs is multithreading. In multithreading system there will be multiple threads running at the same time, to handle various real time tasks. So sharing CPU among the multiple running threads is an important task handled by the operating system. Because of which CPU utilization is a critical factor for any operating system that should be optimized.

The improvement in CPU utilization will result in better performance of any applications running on that operating system. The performance variation can be better visualized in GUI (Graphical User Interface) related applications by the response of the application to the user.

The different RTOs provide various tools to measure the CPU utilization. One of leading RTOs WinCE provides performance monitor tool to measure same.

Now we will continue our discussion with "**How to improve the CPU utilization**?"

The "**Code optimization**" is one of the most efficient ways to optimize the CPU utilization in software. This section will cover simple but powerful C++ points if taken care; a better CPU utilization can be achieved.

The following are 10 points for achieving a better CPU utilization,

1. **Pass objects by reference instead of by value.**

Passing objects by reference is more efficient than passing by value because no new objects are being created. This will reduce the runtime overhead resulting in better CPU utilization.

EXAMPLE:

```
/*Pass object by value*/
Class A
{
 public:
  const A violation( A a )
  {
        return a;
  }
};
/*Pass object by reference*/
Class A
{
 public:
  const A& valid( const A &a )
  {
   return a;
  }
}
```

2. **Prefer initialization to assignment in constructors using initialize list**

Using the initialize list functionality that C++ offers is very important for efficiency. All member objects that are not in the initialize list are by default created by the compiler using their respective default constructors. By calling an object's constructor in the initialize list, you avoid having to call an object's default constructor and the overhead from an assignment operator inside the constructor. Also, using the initialize list may reduce the number of temporaries needed to construct the object.

EXAMPLE:

```
/*Initialization of data members in constructor without initialize list */
#include <string>
using namespace std;
Class A
{
 public:
 A( const char* file, const char* path )
 {
```

```
  myFile = file;
  myPath = path;
 }
 private:
 string myFile;
 string myPath;
};
```

/* Initialization of data members in constructor using initialize list */

```
#include <string>
using namespace std;
class A
{
 public:
 A( const char* file, const char* path ) :
 myFile(file), myPath(path) {} //Initialization list
 private:
 string myFile;
 string myPath;
};
```

3. Avoid use of virtual functions wherever possible

The use of virtual function will affect the performance because of the following reasons,

1. The constructor of an object containing virtual functions must initialize the vptr tabl, which is the table of pointers to its member functions.
2. Virtual functions are called using pointer indirection, which results in a few extra instructions per method invocation as compared to a non-virtual method invocation.
3. Virtual functions whose resolution is only known at run-time cannot be inlined.

4. Avoid slicing function arguments / return value

If the class is having virtual function then the object of that class must be return by reference instead of by value. Passing / returning objects by reference is more efficient than passing by value because no new objects are being created. It also avoids the "slicing problem".

Slicing problem: "Slicing" is where you assign an object of a derived class to an instance of a base class, thereby losing part of the information - some of it is "sliced" away.

EXAMPLE:

/*slicing problem*/

```
Class Pet
{
 public:
```

```
   string name;
};
class Dog : public Pet
{
public:
   string breed;
};

int main()
{
   Dog dog;
   Pet pet;

   dog.name = "Tommy";
   dog.breed = "Kangal Dog";
   pet = dog;
   cout << pet.breed;       //ERROR
}
```

To defeat the problem, use pointers to dynamic variables.

```
/*Solution for the slicing problem*/
Pet *ptrP;
Dog *ptrD;
ptrD = new Dog;
ptrD->name = "Tommy";
ptrD->breed = "Kangal Dog";
ptrP = ptrD;
cout << ((Dog *)ptrP)->breed;
```

5. Do not use user-defined conversion functions

Do not use user-defined conversion functions. User-defined conversion functions will result in creating and destroying temporary objects which will affect runtime. And allows the most time unused code to compile.

User-defined conversion functions: Are often called "cast operators" because they (along with constructors) are the functions called when a cast is used. These functions will provide explicit conversions from a given class type to another type.

Syntax:

```
Conversion-function-name:
  operator conversion-type-name ()
Conversion-type-name:
  type-specifier-list ptr-operator opt
```

The following example specifies a conversion function that converts type Money to type double:

```
//spec1_conversion_functions1.cpp
struct Money
{
  /*User defined conversion function*/
  /*This function is called when a “double” cast is used.*/
 operator double()
 {
  return _amount;
 }
 private:
 double _amount;
};

int main()
{
  Money Account;
  double CashOnHand = Account;

 /*User defined conversion function is called.*/
  cout << (double)Account << endl;

}
```

The initialization of CashOnHand with Account causes a conversion from type Account to type double.

6. Use inline functions wherever possible

Use of inline function is one of the easiest optimizations to be used in C++ and it can result in the most dramatic improvements in execution speed. Because with inline function the compiler insert the complete body of the function in every place the function is being called, rather than generating code to call the function in the place where it is defined, which eliminate the time overhead when a function is called.

7. Consider returning object by reference instead of by value

An object of type class/struct/union should return by reference instead of value. Because returning by value more efficient than returning by value. Sometimes it could be unsafe in some cases. So, the user should check definition of function before the changes, to be sure that a new code is safe and more efficient than previous one.

EXAMPLE:

```
/*Return by value*/
```

```
Class Foo{};

Class A
{
 public:
 Foo foo();
 Foo goo( A& a )
 {
  return foo();
 }
};
/*Return by reference*/
Class A2
{
 public:
 Foo& foo();
 Foo& goo( A2& a )
 {
  return foo();
 }
};
```

8. Constructors allowing for conversion should be made explicit.

Explicit constructors are simply constructors that cannot take part in an implicit conversion. One argument constructors that are not meant for type conversion should be declared explicit. The explicit keyword in C++ is used to declare explicit constructors. It removes the overhead by avoiding the calls to implicit type conversion functions.

EXAMPLE:

```
/*with implicit conversion*/
Class Foo
{
 public:
  Foo(int x);
};

Class Bar
{
 public:
  Bar(double x);
};

void yourCode()
```

```
{
 Foo a = 42;        //OK: calls Foo::Foo(int) passing 42 as an argument
 Foo b(42);         //OK: calls Foo::Foo(int) passing 42 as an argument
 Foo c = Foo(42);    //OK: calls Foo::Foo(int) passing 42 as an argument
 Foo d = (Foo)42;    //OK: calls Foo::Foo(int) passing 42 as an argument

 Bar x = 3.14;       //OK: calls Bar::Bar(double) passing 3.14 as an argument
 Bar y(3.14);        //OK: calls Bar::Bar(double) passing 3.14 as an argument
 Bar z = Bar(3.14);  //OK: calls Bar::Bar(double) passing 3.14 as an argument
 Bar w = (Bar)3.14;  //OK: calls Bar::Bar(double) passing 3.14 as an argument
}
```

/* with explicit conversion */

```
Class Foo
{
 public:
 explicit Foo(int x);
};

Class Bar
{
  public:
    explicit Bar(double x);
};

void yourCode()
{
  Foo a = 42;        //Compile-time error: can't convert 42 to an object of type Foo
  Foo b(42);         //OK: calls Foo::Foo(int) passing 42 as an argument
  Foo c = Foo(42);    //OK: calls Foo::Foo(int) passing 42 as an argument
  Foo d = (Foo)42;    //OK: calls Foo::Foo(int) passing 42 as an argument

  Bar x = 3.14;       //Compile-time error: can't convert 3.14 to an object of type Bar
  Bar y(3.14);        //OK: calls Bar::Bar(double) passing 3.14 as an argument
  Bar z = Bar(3.14);  //OK: calls Bar::Bar(double) passing 3.14 as an argument
  Bar w = (Bar)3.14;  //OK: calls Bar::Bar(double) passing 3.14 as an argument
}
```

9. Use macros wherever possible

Unlike inline functions the macros also eliminate function call overhead as they are always expanded in-line. The macros are more efficient compared to inline functions because inline functions are parsed by the compiler, whereas macros are expanded by the preprocessor.

10. Avoid dynamic memory allocations using new and delete operators

Avoid allocating memory dynamically using new and delete operators. New and delete operators will call the constructor and destructor respectively. This means class instances are initialized and deinitialized automatically. So there's a performance hit compared to plain allocation. To avoid this performance hit use malloc and free functions to allocate memory dynamically. Because malloc and free do not call the constructor and destructor respectively. This means classes won't get initalized or deinitialized automatically.

The above section covered few points (10) to achieve "**code optimization**". In embedded software development there are many C++ tools which can be used to achieve the best possible code optimization. These tools support static (Compile time) as well as dynamic (Run time) code analysis. One of such tool is Parasoft's "**C/C++ Test Tool**".

C/C++ Test Tool: Provides a fully-integrated suite for automating a broad range of best practices proven to improve software development team productivity and software quality. Static code analysis is a feature provided by this tool which will help the developer to optimize the written code.

References:

- http://www.parasoft.com/jsp/products/cpptest.jsp
- http://en.wikipedia.org/wiki/CPU_time

Article By:

Veenadevi.C.Tippimath

Member-Technical

iWave Systems Pvt Ltd

veenac@iwavesystems.com