

Managed C# versus Unmanaged C++

by Randy Charles Morin

This is Howard C-Shell, coming to you live from your typical Windows computer just outside of Disney World. Here, I will find out once and for all how much slower managed C# is then unmanaged C++. In this bout of speed, unmanaged C++ is the clear favorite.

This article is going to run some performance tests against C# and C++. I think it has become generally accepted that Visual C++ is the best, if not, one of the best performance compilers on the Win32 platform. Many developers have approached the dotNet community with questions as to the performance of unmanaged C++ on Win32 versus managed C# in dotNet.

I think I should begin by explaining my development environment. The environmental details are that I've got a Dell Inspiron 3800 G700GT notebook that is slightly over one year old. I have Windows 2000 with SP2, Visual Studio.NET and the dotNet platform.

All the tests are performed with command-line executables that were compiled in release mode and run from the command prompt, not the Visual Studio IDE. These are the results right out of the box. No optimizations. Optimizations to the C++ or the C# compilers might produce different results and this might be a subject for a further article.

I plan on running four different tests. Some of them are classic performance test (Sieves) and others are solely because I want to test the performance of a specific item in the dotNet framework.

- Hello World Test
- Sieve of Eratosthenes
- Database Access Test
- XML Test

An important fact is that I'm trying to make the code in both environments as similar as possible. Please, if you find a discrepancy that favors one language over the other, then please send me some kind words and I'll republish the results.

Hello World

The Hello World test programs measure the amount of time that it takes to load a program and its run-time environment. In the case of C++, that's the C run-time library and pretty lightweight. In C#, the dotNet framework must be loaded, which arguable is not as lightweight.

The code for the C++ Hello World program is presented in Listing 1.

Listing 1: helloworld.cpp

```
#include <iostream>
int main(int argc, char *argv[])
{
    std::cout << "Hello World" << std::endl;
    return 0;
};
```

The code for the C# Hello World program is presented in Listing 2.

Listing 2: helloworld2.cs

```
using System;
namespace HelloWorld
{
    class Class1
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

The results of this test will indicate to us the load time for each environment. A quick load time is often desirable when you have processes that load, perform a very few amount of small tasks and exit. Perl scripts typically had large load times that made them undesirable in performance oriented CGI-base websites. In these cases, C++ was often chosen over similar Perl programs.

In longer-lived programs, the load time can be irrelevant compared to the run-time performance.

The results of running the test 10 times are presented in Table 1.

Table 1: Hello World Test Results

Test Run	C++ (milliseconds)	C# (milliseconds)
1	40	1221
2	20	121
3	10	130
4	10	100
5	10	110
6	10	130
7	10	120
8	10	140
9	10	150
10	20	140
Average	15	235

These results are not accurate to the milliseconds as the `GetTickCount` function is not capable of such accuracy. Rather the results are accurate to about the centisecond (hundredth of a second).

The results reveal a couple of points. First, cold starting a dotNet application produces a more expensive startup than running the same application a second time. Second, the startup cost on subsequent runs is about one tenth of a second longer in dotNet. In most applications, a one-tenth of a second startup cost is irrelevant.

Sieve of Eratosthenes

The Sieve of Eratosthenes test programs measure basic integer arithmetic and comparison logic. The algorithm was devised long before computers and thus is a great proving ground for evaluating human algorithms in various environments.

The C++ code for the Sieve of Eratosthenes program is presented in Listing 3.

Listing 3: sieve.cpp

```
#include <windows.h>
#include <iostream>
#include <algorithm>
#include <vector>
#include <cstdlib>

using namespace std;
int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        std::cerr << "Usage:\tsieve [iterations]\n";
        return 1;
    };

    size_t NUM = atoi(argv[1]);
    DWORD dw = ::GetTickCount();
    vector<char> primes(8192 + 1);
    vector<char>::iterator pbegin = primes.begin();
    vector<char>::iterator begin = pbegin + 2;
    vector<char>::iterator end = primes.end();

    while (NUM--)
    {
        fill(begin, end, 1);
        for (vector<char>::iterator i = begin;
             i < end; ++i)
        {
            if (*i)
            {
                const size_t p = i - pbegin;
                for (vector<char>::iterator k = i + p;
                     k < end; k += p)
                {
                    *k = 0;
                }
            }
        }
    }

    DWORD dw2 = ::GetTickCount();
    std::cout << "Milliseconds = " << dw2-dw
    << std::endl;
    return 0;
}
```

The C# code for the Sieve of Eratostenes program is presented in Listing 4.

Listing 4: sieve.cs

```
using System;
namespace Sieve
{
    class Class1
    {
        static void Main(string[] args)
        {
            if (args.Length != 1)
            {
                Console.WriteLine("Usage:\tsieve "
                    "[iterations]");
                return;
            }

            int NUM = int.Parse(args[0]);
            long dt = DateTime.Now.Ticks;
            int[] primes = new int[8192+1];
            int pbegin = 0;
            int begin = 2;
            int end = 8193;
        }
    }
}
```

```
while (NUM-- != 0)
{
    for (int i = 0; i < end; i++)
    {
        primes[i] = 1;
    }

    for (int i = begin; i < end; ++i)
    {
        if (primes[i] != 0)
        {
            int p = i - pbegin;
            for (int k = i + p; k < end; k += p)
            {
                primes[k] = 0;
            };
        }
    };
};

long dt2 = DateTime.Now.Ticks;
System.Console.WriteLine("Milliseconds = {0}",
    (dt2-dt)/10000);
}
}
```

It's not enough to say that one environment is faster than another. In each test, I'm indicating which language constructs will most impact the results of the test. When you are choosing a language based on the performance, you should be looking at exactly what type of performance you need. In this case, the Sieve of Eratosthenes programs test the loop constructs and both comparison and manipulation of the integer base type.

The results of running the test 10 times with 10000 iterations per test are presented in Table 2.

Table 2: Sieve Test Results

Test Run	C++ (milliseconds)	C# (milliseconds)
1	1342	2724
2	1342	2714
3	1342	2724
4	1342	2724
5	1342	2734
6	1342	2724
7	1362	2734
8	1352	2734
9	1362	2724
10	1352	2724
Average	1348	2726

The results are pretty conclusive. Integer calculations are twice as fast in C++ as C#. That's something to consider. A logic intensive server might have the same issues and might be more suitable as unmanaged C++ than as managed C#.

There is one difference between the C++ and C# code. The C# code uses a native array, whereas the C++ code uses the vector template class. I rewrote the C++ code to use the native array thinking that it would be faster. It wasn't. The native C++ array clocked in the 1900s of milliseconds.

Database Access

In this section, I'll be writing some C++ and C# code that tests both data access and manipulation. All the tests will be against one table with the following data definition.

```
CREATE TABLE testtable
(
  col1 INTEGER,
  col2 VARCHAR(50),

  PRIMARY KEY (col1)
)
```

The test will be divided into three. The first and third tests will focus on data manipulation queries, while the second test will focus on data access queries. The results for data manipulation and data access will be presented separately.

The C++ code for data access and manipulation is presented in Listing 5.

Listing 5: db.cpp

```
#import "msado15.dll" \
no_namespace rename("EOF", "EndOfFile")
#include <iostream>
#include <string>
#include <sstream>

int main(int argc, char* argv[])
{
  if (argc != 2)
  {
    std::cerr << "Usage:\tdb [rows]\n";
    return 1;
  };

  ::CoInitialize(NULL);
  int NUM = atoi(argv[1]);
  DWORD dw = ::GetTickCount();
  _ConnectionPtr conptr(__uuidof(Connection));
  conptr->Open(L"Provider=Microsoft.Jet.OLEDB.4.0;"
    "Data Source=c:\\db.mdb;",
    L"",
    L"",
    adOpenUnspecified);
  for (int i=0;i<NUM;i++)
  {
    VARIANT RecordsEffectd;
    RecordsEffectd.vt = VT_INT;
    std::wstringstream ss;
    ss << L"INSERT INTO testtable (col1, col2) "
      << "VALUES ("
      << i+1 << L", \' " << i+1 << L"\'";
    _bstr_t sql = ss.str().c_str();
    conptr->Execute(sql, &RecordsEffectd, adCmdText);
  };

  DWORD dw2 = ::GetTickCount();
  std::cout << "Milliseconds = " << dw2-dw
    << std::endl;
  dw = ::GetTickCount();
  for (int j=0;j<100;j++)
  {
    _RecordsetPtr rsptr(__uuidof(Recordset));
    rsptr->Open(L"SELECT col1, col2 FROM testtable",
      conptr.GetInterfacePtr(),
      adOpenForwardOnly, adLockOptimistic, adCmdText);
    while (rsptr->EndOfFile)
    {
      _variant_t v1 = rsptr->GetCollect("col1");
```

```
        _variant_t v2 = rsptr->GetCollect("col2");
        rsptr->MoveNext();
    };
    rsptr->Close();
};

dw2 = ::GetTickCount();
std::cout << "Milliseconds = " << dw2-dw
    << std::endl;
dw = ::GetTickCount();
for (int i=0;i<NUM;i++)
{
    std::wstringstream ss;
    VARIANT RecordsEffectted;
    RecordsEffectted.vt = VT_INT;
    ss << L"DELETE FROM testtable WHERE coll = "
        << i+1;
    _bstr_t sql = ss.str().c_str();
    conptr->Execute(sql, &RecordsEffectted, adCmdText);
};
conptr->Close();
dw2 = ::GetTickCount();
std::cout << "Milliseconds = " << dw2-dw
    << std::endl;
::CoUninitialize();
return 0;
}
```

The C# code for data access and manipulation is presented in Listing 6.

Listing 6: db2.cs

```
using System;
using System.Data;
using System.Data.OleDb;
using System.Text;

namespace Db
{
    class Class1
    {
        static void Main(string[] args)
        {
            if (args.Length != 1)
            {
                Console.WriteLine("Usage:\tdb2 [rows]");
                return;
            }

            int NUM = int.Parse(args[0]);
            long dt = DateTime.Now.Ticks;
            OleDbConnection connection;
            connection = new OleDbConnection(
                "Provider=Microsoft.Jet.OLEDB.4.0;"
                "Data Source=c:\\db.mdb");
            connection.Open();
            for (int i=0;i<NUM;i++)
            {
                StringBuilder ss = new StringBuilder();
                ss.Append("INSERT INTO testtable (coll, col2)"
                    " VALUES (");
                ss.Append(i+1);
                ss.Append(", ");
                ss.Append(i+1);
                ss.Append(")");

                OleDbCommand command = new OleDbCommand(
                    ss.ToString(), connection);
                command.ExecuteNonQuery();
            }
        }
    }
}
```

```
long dt2 = DateTime.Now.Ticks;
System.Console.WriteLine("Milliseconds = {0}",
    (dt2-dt)/10000);
dt = DateTime.Now.Ticks;
for (int j=0;j<100;j++)
{
    DataSet dataset = new DataSet();
    OleDbCommand command = new OleDbCommand(
        "SELECT coll, col2 FROM testtable",
        connection);
    OleDbDataReader reader =
        command.ExecuteReader();
    while (reader.Read() == true)
    {
        int v1 = reader.GetInt32(0);
        string v2 = reader.GetString(1);
    };
    reader.Close();
};

dt2 = DateTime.Now.Ticks;
System.Console.WriteLine("Milliseconds = {0}",
    (dt2-dt)/10000);
dt = DateTime.Now.Ticks;
for (int i=0;i<NUM;i++)
{
    StringBuilder ss = new StringBuilder();
    ss.Append("DELETE FROM testtable "
        "WHERE coll = ");
    ss.Append(i+1);
    OleDbCommand command = new OleDbCommand(
        ss.ToString(), connection);
    command.ExecuteNonQuery();
};

connection.Close();
dt2 = DateTime.Now.Ticks;
System.Console.WriteLine("Milliseconds = {0}",
    (dt2-dt)/10000);
}
}
```

The results of running the test 10 times with 100 rows per test are presented in Table 3.

Table 3: Database Test Results

Test Run	C++ (milliseconds)	C# (milliseconds)
1	1612/441/450	4086/630/560
2	391/410/441	490/630/520
3	370/421/440	480/510/440
4	371/420/451	470/510/450
5	370/421/461	460/500/450
6	371/420/461	470/500/460
7	370/411/471	470/500/460
8	381/410/451	460/510/470
9	370/421/450	470/510/470
10	391/410/461	460/510/470
Average	499/419/454	832/531/475

The results were quite surprising to me. Considering the benefits you get with dotNET, I don't think it's much to expect a 25% decrease in performance. I think dotNET is a winner here.

XML

The latest craze in computing is XML. Many will be interested in the C# XML parsing performance versus the same performance on Visual C++.

The C++ code for xml access and manipulation is presented in Listing 7.

Listing 7: xml.cpp

```
#import <msxml3.dll> named_guids
#include <iostream>
int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        std::cerr << "Usage:\txml [filename]\n";
        return 1;
    };

    ::CoInitialize(NULL);
    DWORD dw = ::GetTickCount();
    for (int i=0;i<100;i++)
    {
        MSXML2::IXMLDOMDocumentPtr DomDocument(
            MSXML2::CLSID_DOMDocument) ;
        _bstr_t filename = argv[1];
        DomDocument->async = false;
        DomDocument->load(filename);
    }

    DWORD dw2 = ::GetTickCount();
    std::cout << "Milliseconds = " << dw2-dw
        << std::endl;
    ::CoUninitialize();
    return 0;
}
```

The C# code for xml access and manipulation is presented in Listing 8.

Listing 8: xml.cs

```
using System;
using System.Xml;

namespace xml2
{
    class Class1
    {
        static void Main(string[] args)
        {
            if (args.Length != 1)
            {
                Console.WriteLine("Usage:\txml [filename]");
                return;
            }

            long dt = DateTime.Now.Ticks;
            for (int i=0;i<100;i++)
            {
                XmlDocument doc = new XmlDocument();
                doc.Load(args[0]);
            }

            long dt2 = DateTime.Now.Ticks;
            System.Console.WriteLine("Milliseconds = {0}", (dt2-dt)/10000);
        }
    }
}
```

The results of running the test 10 times are presented in Table 4.

Table 4: XML Test Results

Test Run	C++ (milliseconds)	C# (milliseconds)
1	241	1111
2	170	841
3	161	841
4	170	861
5	160	861
6	171	851
7	170	841
8	160	831
9	160	841
10	170	851
Average	203	873

These results were again very surprising to me. It's hard to believe that the dotNET XML classes are four to five times slower than equivalent ActiveX classes. It may be that under the hood, the dotNET classes are doing something different that will save me time somewhere else. Otherwise, I hope Microsoft will spend some time optimizing their dotNET XML classes.

Conclusion

Something that must be remembered is that the dotNET framework is newer than all the technologies that I measured it against today. As such, there should be a lot of room for optimizations within the framework. What also must be said is that I've only started performance testing dotNET. I could only fit four tests into a brief article and there are obviously many other components that might be faster or slower with dotNET.

About the Author

Randy Charles Morin is the Chief Architect of 1X Inc [www.1xinc.com] from Toronto, Ontario, Canada and lives with his wife and two kids in Brampton, Ontario. He is the author of the www.kbcafe.com website, many articles and many books.