W H I T E   P A P E R

.NET interop and performance analysis

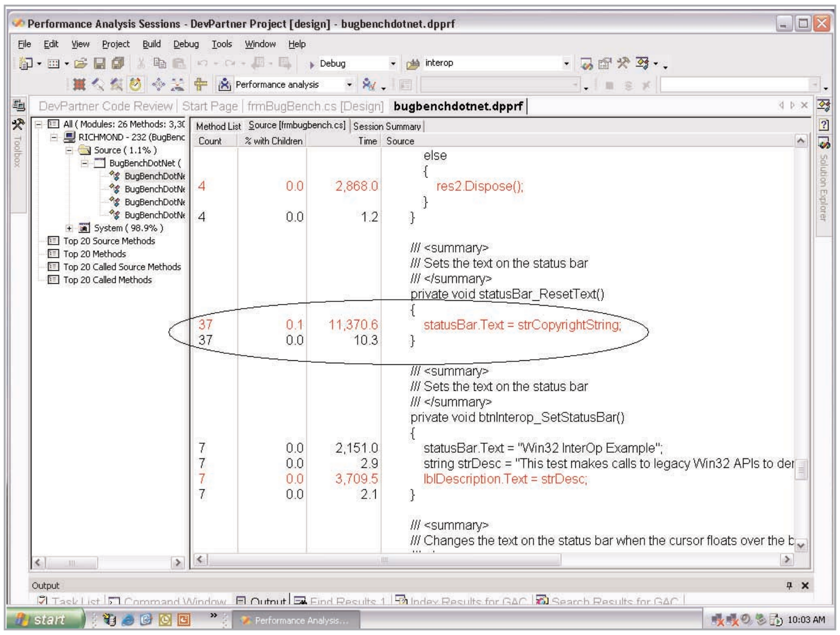
Calling native code from your .NET application has performance implications. Use the right method to call your native code and be ready to measure its performance when you do.

With Microsoft’s introduction of the .NET platform and languages, including its bias toward distributed application components, performance analysis and performance tuning have become substantially more important for today’s development efforts. Individual assemblies and web services that seem to offer adequate performance when unit tested perform unacceptably when integrated as a single application.

One of the biggest, yet largely unexplored, areas of .NET performance concerns the so-called interop—the interoperation of managed and unmanaged code in the same application. In most cases, this involves new .NET code calling native code components. However, it’s also possible for native applications to call .NET components, although by its nature this is likely to be much less popular.

CompuwareCorporation

At this point, many developers don't understand the performance implications of interop. Moreover, developers aren't necessarily even cognizant of when their applications perform interop, and what they can do to resolve problems with it. In some cases, interop is performed by the .NET Framework and most developers think it can't be helped. For example, Figure 1 shows the negative performance implications of a line of code that calls into native code indirectly through its children.


Performance Analysis Sessions - DevPartner Project [design] - bugbenchdotnet.dpprf
File Edit View Project Build Debug Tools Window Help
Debug
interop
Performance analysis
DevPartner Code Review | Start Page | frmBugBench.cs [Design] | bugbenchdotnet.dpprf
All (Modules: 26 Methods: 3,3
RICHMOND - 232 (BugBenc
Source (1.1%)
BugBenchDotNet (
BugBenchDotN
BugBenchDotN
BugBenchDotN
BugBenchDotN
System (98.9%)
Top 20 Source Methods
Top 20 Methods
Top 20 Called Source Methods
Top 20 Called Methods
Method List | Source [frmbugbench.cs] | Session Summary
Output
Task List | Command Window | Output | Find Results 1 | Index Results for GAC | Search Results for GAC
start
Performance Analysis...
10:03 AM

```
Count   % with Children      Time   Source
                                           else
                                           {
4                   0.0   2,868.0            res2.Dispose();
                                           }
4                   0.0       1.2      }

                                       /// <summary>
                                       /// Sets the text on the status bar
                                       /// </summary>
                                       private void statusBar_ResetText()
                                       {
37                  0.1  11,370.6        statusBar.Text = strCopyrightString;
37                  0.0      10.3      }
                                       /// <summary>
                                       /// Sets the text on the status bar
                                       /// </summary>
                                       private void btnInterop_SetStatusBar()
                                       {
7                   0.0   2,151.0        statusBar.Text = "Win32 InterOp Example";
7                   0.0       2.9        string strDesc = "This test makes calls to legacy Win32 APIs to der
7                   0.0   3,709.5        lblDescription.Text = strDesc;
7                   0.0       2.1      }

                                       /// <summary>
                                       /// Changes the text on the status bar when the cursor floats over the b
```

Figure 1. One of the most computationally expensive lines of code is one that calls native code components.

In general, there are three reasons why you might want to call unmanaged code from a .NET application. The first is that you have no choice in the matter. You have a third-party control or component, and a .NET equivalent isn't available. So, you don't have source code and you still have to use the component even though the rest of your application is now running as managed code.

Second, you might call unmanaged components because you have working code that you haven't yet ported into a managed language. In fact, you may not even have any plans to port some code. For instance, you may have invested a large amount of effort in COM components or legacy code that encapsulate unchanging business logic, and it adds no value to rewrite the code at this time. So you continue to use the same unmanaged code even though the rest of the application is new.

The third reason for calling unmanaged components is that no way can be found within the .NET Framework to do what the application needs. In this case, you might have to make a call directly to the Windows platform or to other third-party libraries, to get the type of event or resource you're looking for.

The problem with mixing managed and unmanaged code in a single application is that the unmanaged or native code isn't recognized in the .NET environment. Managed code components not only depend on the classes available in the .NET Framework, but they also expect the other components with which they interact to depend on that framework as well.

Nonetheless, you've made the decision to call a native component or code from your .NET application. Before you do so, you have to be cognizant of what it involves and what the performance implications can be. All but the most trivial calls into native code must undergo a mode transition. This is the physical process of moving data between the managed and unmanaged modes of operation, typically requiring about two dozen instructions. The second cost is marshalling the data to move across the boundary. Marshalling is necessary because the internal representation of data structures is different between managed and unmanaged code. To pass data across the boundary, you have to change the data from the .NET representation to the native one, and then back again.

Now here's the rub. Marshalling is computationally expensive, and the more data you move back and forth, the more expensive it becomes. Marshalling data structures one way can add as much as 3,000 instructions to your processing time for complex data.

There are five ways to transition code from managed to unmanaged. These ways can be grouped into two categories: COM-based and traditional DLLs. Traditional DLLs have three ways to be hooked into a managed program. One, an *internal call*, is a mechanism that is limited to small bits of unmanaged code, typically under 100 instructions. With this method no mode transition occurs and no marshalling is allowed, and only the default data structure translators will be used. This is similar to inline assembly code in a C program. In effect, this code is linked into the managed executable and executed as managed code.

Second, is the *IJW (It Just Works) method*. This method statically links the called library into the managed executable. A mode transition occurs but no marshalling is allowed. The Visual C++ compiler does all the work, so there's nothing that needs to be done on your part. Note that you can't pass more than the simplest data across the boundary.

Third, is the *Platform Invoke (P/Invoke) method*, which is by far the most popular way of mixing managed and unmanaged code into a .NET application. This method consists of a mode transition and data marshalling, so you have a lot more flexibility in what you call and what data you pass.

The advantages and perils of COM interop

The latter category of transitions focuses on COM access. These transitions make a lot of sense from a practical standpoint. Microsoft is encouraging Windows developers to start building new applications in Visual Studio .NET, yet you're not going to throw away perfectly good code in your existing applications to do so. In some cases, you may be able to migrate your code to one or more .NET languages along with the rest of the application. In many cases, however, what you're going to do is take the existing logic, often encapsulated in COM/COM+ components, and use it "as is" with the new Microsoft .NET code.

Microsoft makes this possible, but with the risk of introducing an entirely new class of potential performance issues into your applications. Finding and fixing these errors is crucial because errors in your native code can have a subtle, yet significant, impact on the managed code in your application. These errors are entirely unlike those you are likely to encounter working within the .NET Framework, and they are largely undetectable with the new and emerging .NET development tools from Microsoft and others.

There are two mechanisms that you can potentially use to call COM objects from .NET code. Both of these mechanisms work in the same way, so depending on which one you use, you can determine what .NET language you're working in and whether or not the COM component is being shared among multiple applications.

The way out of this dilemma is to use a piece of code that acts as a proxy to the unmanaged COM component. The type of proxy used in calling unmanaged code from managed code is known as a Runtime Callable Wrapper (RCW). This wrapper is needed to work with COM-type libraries, which contain metadata describing the public interface to COM components. The job of an RCW is to convert the existing COM metadata to .NET metadata, which is readable by managed application components.

There are two ways to convert COM component metadata into a form usable by .NET. One tool for performing this conversion is called *tlbimp*, which is part of the .NET Framework Software Developer Kit (SDK). Tlbimp reads the metadata from a COM type library and creates a CLR assembly incorporating the metadata for calling the COM component.

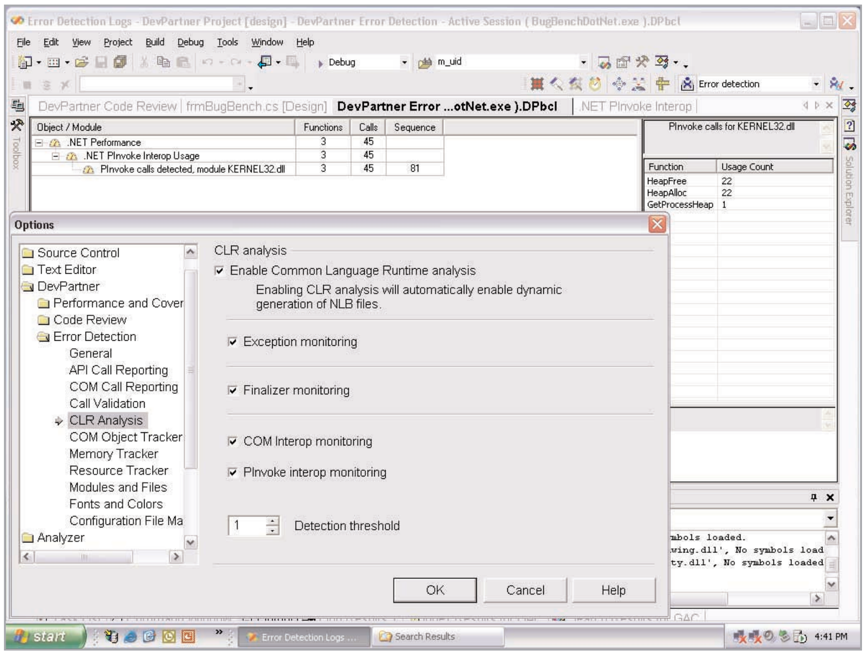
The second approach is to call the COM component directly. This option is available only when calling from Visual Basic .NET code. For a Visual Basic .NET project, all you have to do is add the COM component from the Add Reference menu selection, which will automatically create the RCWs for the selected type libraries in your project. This operation results in the creation of a DLL with a name derived from the original COM component name.

The major drawback to this second alternative is that there's no opportunity to sign the resulting code in order to place it in the Global Assembly Cache. As a result, the component can't be shared among multiple .NET applications. It's considered to be a private component, usable only from within your Visual Basic .NET application. You must also use the first approach if you require certain details of the component, such as its version number.

Either approach can be a computationally expensive proposition. Both still require the mode transition as well as marshalling of data. Don't count on being able to automatically generate the RCW to get better performance. The alternative development techniques are for convenience rather than advantage.

Preparing to use interop

The first step in addressing the performance issues surrounding interop is understanding what it is costing you. You can do this with a performance analysis tool such as Compuware DevPartner Studio, which measures the amount of time taken to perform a mode transition and marshalling as well as the number of P/Invoke and Runtime Callable Wrapper calls made. Your first focus should be on these types of calls, as they require marshalling and can be a significant performance drag on the overall application.


Error Detection Logs - DevPartner Project [design] - DevPartner Error Detection - Active Session (BugBenchDotNet.exe).DPbcl

Object / Module	Functions	Calls	Sequence
.NET Performance	3	45	
.NET PInvoke Interop Usage	3	45	
PInvoke calls detected, module KERNEL32.dll	3	45	81

PInvoke calls for KERNEL32.dll

Function	Usage Count
HeapFree	22
HeapAlloc	22
GetProcessHeap	1

Options

CLR analysis

☑ Enable Common Language Runtime analysis

Enabling CLR analysis will automatically enable dynamic generation of NLB files.

☑ Exception monitoring

☑ Finalizer monitoring

☑ COM Interop monitoring

☑ PInvoke interop monitoring

1 Detection threshold

```
mbols loaded.
wing.dll', No symbols load
ty.dll', No symbols loaded
```

Figure 2. Compuware DevPartner Studio shows the number and type of COM Interop calls from managed to unmanaged code.

Once you have this information, you can determine if the performance of your application is acceptable or if you need to make changes to speed up the mode transitions. If those calls appear to take an excessive amount of time in comparison to the rest of the application, you can focus on improving their performance.

One type of change you can consider is whether your managed/unmanaged interface should be “chatty” or “chunky.” As the names imply, chatty calls are those that occur often and pass little data, while chunky calls occur less frequently, but do more work when they do occur. While at first glance, it might appear that chunky calls are more efficient, because you are doing the mode transition less frequently, that’s not necessarily the case. A chatty interface passing less complex data more often may turn out to be less computationally expensive because its marshalling isn’t as complex.

How do you determine whether or not you should be using chatty or chunky mode transitions? There’s no easy way that applies to all circumstances; it depends on the amount of data and frequency of calls. The best thing to do is to prototype each type of interface and profile its performance. By investing a little time early in the development phase, you can ensure that you made the right performance choice and not have to go back and make substantial changes after you have a working application.

That’s not to say you might not have to go back and make adjustments to your calls once the application is done. For example, you may find that in certain parts of the application, chunky calls are more efficient because of the overall volume of calls. But prototyping your data and calls ahead of time gives you the ability to make better decisions.

Chances are you will be doing interop with native code from your .NET applications on a number of occasions over the next several years. Understanding the alternatives that are available, when to use them and how to evaluate their performance will make it possible for you to call native code with confidence.

COMPUWARE®

www.compuware.com

04/03