Welcome to the CLR Code Generation Team's blog

September 15, 2007, 3:18 am

≫ Next: To NGen or Not to NGen?

This is the first blog post from the code generation feature team working on the Common Language Runtime (CLR). We're the group of individuals that make it possible to generate native code for all binaries that run on top of the Microsoft .NET Framework. Since all managed applications today are distributed in a format known as MSIL (for Microsoft Intermediate Language), the CLR is responsible for compiling MSIL to directly-executable machine code. That's where we come in -- currently we support compiling against 3 different machine architectures -- x86, x64, & IA64, and in 2 different compilation modes -- dynamic compilation using the Just-in-time compiler (JIT), and ahead-of-time/pre-compilation using NGen (for Native Generation). Two examples of design challenges in our space include balancing the quality of the generated code with the time taken to generate it, and balancing the desire to update NGen images soon after MSIL binaries are patched with the desire to keep patch install time at a minimum.

Expect posts about our JIT & NGen back-ends over the coming months on this blog. Also, please feel free to let us know what codegen-related topics you're interested in reading about.

Image Not Found

To NGen or Not to NGen?

September 15, 2007, 3:19 am

≫ Next: Running NGen as part of installing a Microsoft Exchange patch roll up takes ~2 ho…
≪ Previous: Welcome to the CLR Code Generation Team's blog

One of the topics we often get questions on is about when it makes sense to invest the extra effort to pre-compile assemblies via NGen instead of simply relying on the JIT compiler to generate native code on the fly at application runtime. I thought I would try to answer that question in our first "real" post.

The JIT is optimized to balance code generation time against code quality, and it works well for many scenarios. NGen is essentially a performance optimization; so just like for anything else that's performance-related, you'll want to measure performance with vs. without NGen, and then decide whether it really helps your specific application and scenario. Here are a few general guidelines.

When to use NGen:

- Large applications: Applications that run a lot of managed code at start up are likely to see wins in start up time when using NGen. Microsoft Expression Blend for example, uses NGen to minimize start up time. If a large amount of code needs to be JIT-compiled at start up, the time needed to compile the IL might be a substantial portion of the total app launch time (even for cold start up) . Eliminating JIT-compilation from start up can therefore result in warm as well as cold start up wins.
- Frameworks, libraries, and other reusable components: Code produced by our JITs cannot be shared across multiple processes; NGen images, on the other hand, can be. Therefore any code that is likely to be used by multiple applications at the same time is likely to consume less memory when pre-compiled via NGen. Almost the entire Microsoft .NET Framework for instance, uses NGen. Also Microsoft Exchange Server pre-compiles its core DLLs that are shared across various Exchange services.
- Applications running in terminal server environments: Once again NGen helps in such a scenario because the generated code can be shared across the different instances of the application – that in turn increases the number of simultaneous user sessions the server can support.

When not to use NGen:

- Small applications: Small utilities like caspol.exe in the .NET Framework aren't NG because the time spent JIT-compiling the code is typically a small portion of the o start up time. As a matter of fact, since NGen images are substantially bigger in si the corresponding IL assemblies, using NGen might actually result in increased di I/O and hurt cold start up time.
- Server applications: Server applications that aren't sensitive to long start up times don't have shared components are unlikely to benefit significantly from NGen. Ir cases, the cost of using NGen (more on this below) may not be worth the benefit C

If it seems that y
keep in mind:

1. N
   F
   h
   b
   Note
2. J
   s
   w
   g
   i
   h
3. U
   issue NGen commands to ensure that the images invalidated by the patch get regenerated. .NET Framework 2.0 supports an "ngen.exe update /queue" comma should work well in most cases (this will regenerate all invalid NGen images at m idle time). If specific assemblies are perf-critical and need to have up-to-date NG images immediately after the patch install, you can issue separate NGen commar those ("ngen.exe install <CriticalDLL>"), followed by the "ngen.exe update /queu command.
4. Using NGen implies your application's disk footprint will increase (NGen images a big when compared to MSIL assemblies).
5. In a handful of scenarios, cold start up and/or application execution can get slowe you use NGen, so you'll definitely want to measure the performance with vs. witl NGen for all scenarios, and validate that it's a win overall for your application to b NGen.

Finally, another important thing to keep in mind is that NGen isn't a magic solution for applicatio time. NGen can speed up application start up by eliminating the time needed for JIT compiling th executed at start up.

However, regardless of whether you do or don't use NGen, it is important to ensure that your ap is architected to be performant. So for instance, if you care about start up time, you should try tc minimize the amount of code that needs to get loaded and executed on your application's start

Below are some good resources on measuring and optimizing application start up performance:
http://blogs.msdn.com/vancem/archive/tags/Perf/default.aspx
http://msdn.microsoft.com/msdnmag/issues/06/02/CLRInsideOut
http://msdn.microsoft.com/msdnmag/issues/06/03/WindowsFormsPerformance/

And some for NGen:
NGen internals: http://msdn.microsoft.com/msdnmag/issues/05/04/NGen/default.aspx
NGen Service (NGen-ing your assemblies in the background):
http://blogs.msdn.com/davidnotario/ (David isn't on our team any longer)

Thanks for visiting our blog!

Running NGen as part of installing a Microsoft Exchange patch roll up takes ~2 hours [Lakshan Fernando]

October 3, 2007, 12:04 pm

≫ Next: How to see the Assembly code generated by the JIT using Visual Studio
≪ Previous: To NGen or Not to NGen?

I work in the CodeGen test team and wanted to share a recent customer experience that was related to ngen. One of our Customer Service and Support (CSS) engineers in France contacted us regarding an installation delay with the latest Microsoft Exchange Server 2007 update rollup. Apparently the patch installer was spending 2 hours generating up-to-date NGen images.

The exchange installer package issues an “ngen update” command at the very end that causes all Exchange assemblies affected by the update to be recompiled synchronously. There is a known issue with this approach that had affected some customers whose machines had been updated with a .NET Framework 2.0 patch just prior to installing the Exchange patch. The .NET Framework 2.0 patch issues an “ngen.exe update /queue” command which schedules background assembly compilation at machine idle time. Then when the synchronous NGen command is issued by the Exchange installer package, it triggers compilation of all managed assemblies that don’t have up-to-date NGen images, including the .NET Framework ones. We originally thought that this might have been the reason behind the long time taken to regenerate the NGen images on this customer’s machine too. However, that was ruled out by the CSS engineer after further investigation.

We then requested the CSS engineer for the two ngen log files created by ngen and its service found at the .Net Framework redistributable installation directory (in this case, ngen.log and ngen_service.log found at %WINDIR%\Microsoft.NET\Framework64\v2.0.50727) and found an interesting anomaly. The ngen compilation worker might sometimes encounter an assembly that is already up-to-date, and the time taken to perform this validation is typically less than a second (the compilation worker exits as soon as it finds that the assembly already has a valid NGen image). But in this customer’s case, each of these assemblies was taking 10-14 seconds to determine that they already had a valid image (see below). These extra 10-14 seconds when multiplied across many different assemblies was resulting in a significant increase in patch install time.

08/24/2007 **10:36:21** [5660]: Compiling assembly **Microsoft.Exchange.Data.Common, Version=8.0.681.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35** ...

08/24/2007 **10:36:33** [5660]: Assembly **Microsoft.Exchange.Data.Common, Version=8.0.6 Culture=neutral, PublicKeyToken=31bf3856ad364e35** is **up to date**.

Around the same time, one of our developers was helping out on a Certificate Revocation L issue report by Policies" (http:/ customer's mac for publisher's customer's mac up the certificat firewall and did assembly. This expected 10 mi IE since the ma

Short summary in the patch inst machine idle-tim commands for p

How to see the Assembly code generated by the JIT using Visual Studio

October 19, 2007, 4:46 pm

≫ Next: How are value types implemented in the 32-bit CLR? What has been done to impr…
≪ Previous: Running NGen as part of installing a Microsoft Exchange patch roll up takes ~…

by Brian Sullivan

In Visual Studio you can set a breakpoint at any line in your source code. When you run your program Visual Studio will break and stop execution when it reaches your breakpoint. At this point you can right click on your source code and select **Go To Disassembly**. You will see the assembly language instructions that were created by the JIT compiler for this method.

The JIT compiler generates either Debug code or Optimized code

If this is your first time looking at the JIT assembly code you undoubtedly are looking at the Debug code generated by the JIT compiler.

This Debug code is not the high quality optimized code that the JIT compiler can generate.

Instead it is basic assembly code that does not have any optimizations applied to it. All local variables references are references into the local stack frame storage because in Debug code the JIT does not enregister any local variables. Another property of Debug code is that **nop** instructions are inserted before some source code statements.

You probably don't want to spend much time looking at the Debug JIT code. Instead you probably want to look at Optimized JIT code. By looking at Optimized JIT code you can confirm that the assembly code for your important methods are well optimized. And if they are not optimized as well as you want, you can experiment to see if making some source code changes can improve the Optimized JIT code.

There are several settings that you will need to change in order for you to see the Optimized code generated by the JIT compiler.

1. The default configuration is **Debug** and you want to select the **Release** configuration.
    - Select the **Release** configuration
2. Make sure that PDB files get created for Release builds.
    - Set **Generate debug info** topdb-only.

More information on why this is necessary:

Ideally you would have wanted it to be the case that simply changing the configuration in the Solution Configuration window to 'Release' would be sufficient. However by default the 'Release' builds do NOT build a program data base (PDB) file for your program. The (PDB) file is essential when using a debugger as it holds the mapping of the source code lines and local variables for your program.

To fix this go to the properties for the project (right click on the project in the Solution Exp
select the 'Build' tab and click the 'Advanced' button all the way at the bottom (you may r
scroll to see it). In the dialog box that comes up there will be a line 'Debug Info'. It shoul
to 'pdb-only'. This tells the compiler to still compile with optimizations, but to also create
file.

3. Configure the Debugging Options in Visual Studio to allow the JIT to generate optimized c
to allow

Go to T

-
-

More in

The Visu

Whenever you launch a managed program under Visual Studio using (Start-Debugging or I
by default, force the JIT to create Debug code. This is true even when you have selected
'Release' configuration. The reason for this is to improve the debugging experience, but it a
it impossible to see the Optimized code that you will get whenever your program is not run
under the Visual Studio debugger. Your alternative of launching the program using (Start W
Debugging or Ctrl+F5) does allow the JIT to create optimized code but Visual Studio wor
of your break points in your code. Thus you normally won't be able to stop and examine t
assembly code for a method. We actually want do some debugging with the Optimized JI

Another problem is that Visual Studio 2005 has a new feature called 'Just My Code' in wh
debugger will not step into any code that it does not believe is being developed. Instead it
over the code. This is also to improve the debugging experience, as most typical users do
to step into Microsoft code such as the Collections classes, etc… However any optimized
also not considered as 'Just My Code' so it too will be step over and is skipped. Again thi
it impossible to actually stop and see the optimized code.

Note that these are global settings as they affect all solutions; however I have found I don't miss ar
'features'. That is because any code that is compiled for the 'Debug' configuration is still not optin
the JIT so you will still get good debugging there.[*]

In my next blog entry I will demonstrate some of the optimizations that we perform in the JIT comp

[*] Note that much of this article was adapted from an earlier article by Vance Morrison (What does forea actually do?)

How are value types implemented in the 32-bit CLR? What has been done to improve their performance?

November 2, 2007, 4:24 pm

≫ Next: Performance implications of unmanaged array accesses

≪ Previous: How to see the Assembly code generated by the JIT using Visual Studio

By Fei Chen

How are value types implemented in the 32-bit CLR?

Value types are the closest thing in the common language runtime model to C++ structures. An instance of a value type is simply a blob of data in memory that contains all the fields in the instance. The main difference between an instance of a value type and an instance of a reference type is that the former does not contain the type ID in its blob (see the example below), because the type information for value types is only needed at compile time.

```
struct PointStruct()  { int x; int y; }  // The memory needed for the instance of this value type is 8 bytes, with 4 bytes for each integer field.

class PointClass()  { int x; int y; }  // The memory need for the instance of this reference type is 12 bytes, with the first 4 bytes containing the type ID, followed by 4 bytes for each integer field.
```

Being a contiguous blob of data in memory, value type instances are referenced internally in the CLR using the pointer to the beginning of the blob of memory.

A value type instance can live in one of two different places – a value type local variable, or a value type field in a reference type[1]. When a value type local variable is declared, the prolog of the jitted code reserves a piece of stack memory[2] (large enough to hold the instance of this value type), and the pointer to this stack location is used in all the places in the jitted code where this local variable is referenced. In the case of a value type field embedded in a reference type, the memory on the heap for this object contains the memory needed for its value type field. See this example:

```
class PointWithColorClass()  { int  color; PointStruct point; }
```

The size of the above object is 16 bytes. The first 4 bytes is the type ID; the next 4 bytes is the *color* field; and the last 8 bytes is for the *point* value type field.

The pointer to the beginning of *point* field is used in all the places in the jitted code where is referenced. (Note that this time the pointer points to a location on the heap, instead of t stack.)

Common operations on value types

There are only a internally.

- Field ac

  Given th accessi offset. I a “mov writes t

- Initializa

  Zero init memory

- Assignment

  Assignment from an instance of a value type to another is done by calling memcpy l these two pieces of memory.

- Calling the instance method

  CLR supports calling instance methods on value types. This is internally done by pa pointer to the instance as a first parameter to the target method. This should sound to people who are familiar with C++ instance method calls, where the “this” pointe passed as the first parameter.
  Since JIT owns the code generation for both the caller and the callee methods, it kn to generate correct code for the value type instance method. In other words, it expe first parameter to be the pointer to the blob of the value type instance.

- Passing as an argument by-value

  Passing a value type instance as a by-value argument requires making a stack copy passing the pointer to this copy to the target method. Consider what needs to be dc call site of Foo() in the following example:

  ```
  static void Foo(int i, string s, PointStruct pointArg) { … }
  static void Main() { PointStruct point; Foo(1, “one”, point); }
  ```

  What happens at the call site can be described using the following pseudo code in syntax:

  ```
  PointStruct stackCopyOfPoint; // This is a stack local variable.
  stackCopyOfPoint = point;  // (or think of it this way) memcpy(&stackCopyOfPoi
  &point, 8);
  Foo(1, “one”, &stackCopyOfPoint);
  ```

  The stack copy is necessary for maintaining the by-value semantics so the callee or the copy and hence has no way to affect the original one.

- Passing as an argument by-reference

  Passing a value type instance as a by-reference argument is easy. Just pass the poi the callee and the caller see the same instance. So any change done inside the call affect the caller.

- Returning a value type

  Returning a value type requires the caller to provide the storage. The pointer of the storage buffer is then passed in as a hidden parameter to the callee. The callee is responsible of filling in this buffer. Consider this example,

  ```
  static PointStruct Bar() { … }
  static void Main() { Bar(); }
  ```

  What actually happens at the call site can be described using the following pseudo

  ```
  PointStruct tempPoint; // A temporary stack local created to hold the return valu
  Bar(&tempPoint);
  ```

  In the case of an embedded value type field, consider this example:

  ```
  static PointStruct Bar() { … }
  static void Main() { PointWithColorClass obj; obj.point = Bar(); }
  ```

  What actually happens at the call site is:

  ```
  Bar(&(obj.point));
  ```

Inefficiencies in the code generation with regards to value types in .NET 2.(

Code generation for value types in .NET 2.0 has several inefficiencies.

1) All value type local variables live entirely on the stack.
2) No assertion propagation optimization is ever performed on value type local variabl
3) Methods with value type arguments, local variables, or return values are never inlin

While the original intent of supporting value types in the CLR was to provide a means for c “lightweight” objects, the actual inefficiencies in the code generation make these “lightwe objects not-so-light.

For bullet 1), the following code would mean 3 stack operations, one for each field access:

```
static void MyMethod1(int v) {
    PointStruct point;  // point will get a stack location.
    point.x=v; point.y=v*2;
    Console.WriteLine(point.x); // All 3 field accesses involve stack operations.
```

}

Wouldn't it be nice if the jitted code stored both fields of *point* into registers and avoided a stack space for this value type local variable altogether?

For bullet 2), the following code would mean 19 useless memcpy's.

```
static void MyMethod2() { PointStruct point1, point2, ..., point20;  point1.x = point1.y = 5;
    point2 = point1; point3 = point2;  ...  point20 = point19;
    Console.
```

Wouldn't it be r
morph the abov

For bullet 3), a

```
struct PointS
static void M
    Console.
```

Currently the JI
have been take

Improving va

Improving code generation with regards to value types has always been a top customer ask
according to MS Connect: http://connect.microsoft.com/VisualStudio/feedback/ViewFeedba
FeedbackID=93858.

Over the past year or so, the JIT team has been working on significant improvements to val
code generation, as well as the inlining algorithm. In summary, all of the above limitations
being eliminated.

The new inliner will allow inlining methods with value type arguments, local variables or re
value. This solves the issue in bullet 3).

An algorithm called "value type scalar replacement" has been implemented to address the
in bullets 1) and 2). This algorithm is based on the observation that a value type local varia
logically can be viewed as a group of independent local scalars, each representing a field i
value type local variable, if

a) there is no operation in the current method that causes any interaction between the fields;

and

b) The address of this value type local variable is never exposed outside of the current

When the above conditions are met, the MyMethod1() listed above can be safely transform

```
static void MyMethod1(int v) {
    int x; int y;  // Was "PointStruct point;". Now replaced by x and y.
    x=v; y=v*2; Console.WriteLine(x);
}
```

by replacing the value type local variable *point* with a group of independent integer local va
namely x and y.

And the MyMethod2() listed above will be transformed to

```
static void MyMethod2() { int x1, y1, x2, y2, ..., x20, y20;   x1 = y1 = 5;
    x2 = x1; y2 = y1; x3 = x2; y3 = y2; ...  x20 = x19; y20 = y19;
    Console.WriteLine(x20 + y20); }
```

Furthermore, the assertion propagation algorithm and the constant folding algorithm will be
to these scalars, since none of them have their address taken. As a result, the code will be
to:

```
static void MyMethod1(int v) { Console.WriteLine(v); }
static void MyMethod2() { Console.WriteLine(10); }
```

In addition, the register allocation algorithm will home the local variable *v* into a machine r
so no stack operation will occur in MyMethod1().

Not all value type local variables can be replaced by scalars, however. Local variables with
address taken, and exposed outside of the current method, cannot be replaced. Consider th
example where SomeBigMethod() is an instance method in PointStruct that is not inlined.

```
static void MyMethod4() { PointStruct point;  point.SomeBigMethod(); }
```

The address of *point* is taken and passed as the "this" pointer to SomeBigMethod(). What
SomeBigMethod() does with this pointer is totally out of the control of MyMethod4(). In this
point is not replaced by scalars. Another way to expose the address of a value type local va
to pass it as a by-reference argument to another method. Taking the address of a value typ
variable and storing it in a static variable, or in an object, also exposes the address.

The JIT in CLR v.Next will be able to perform value type scalar replacement optimization on
following kinds of value types whenever it thinks it will be beneficial:

1) The value type contains no more than 4 fields.
2) The types of the fields in the value type are either primitive types or object referenc
3) The value type must be using [StructLayout(LayoutKind.Sequential)], which is the de

Guidelines forusing value types in the CLR

The decision around whether to use value types, or not, should be based primarily on the s
of the program. Value types should be used when the pass-by-value semantics are the mos
and the most frequently used in the program.

After the decision has been made to use the value type, it is time to think about the perfor
implications, and to determine how to help the JIT generate the best possible code. Always

mind that the by-value nature of value types means that a lot of copy operations might be happening under the covers. Also, nearly every operation related to a value type will be a n operation (either operated on the stack or on the heap) if this value type is not replaced by

Developers should examine the jitted code of their hot methods under the debugger to mak the value type stack local variables are indeed homed in registers.

Try not to creat type instance m exposed. When rather than the scalars.

[1] For simplicity, I
[2] This is not true described later in

Performance implications of unmanaged array accesses

February 8, 2008, 6:13 pm

>> Next: What's in NetFX 3.5 SP1?
<< Previous: How are value types implemented in the 32-bit CLR? What has been done to i…

I was recently shown the following code and asked why the loop calling SafeAccess executed significantly faster than the second loop calling UnsafeAccess:

```
static int [] intarray = new int [5000];

static void SafeAccess(int a, int b)
{
    int temp = intarray[a];
    intarray[a] = intarray[b];
    intarray[b] = temp;
}
static Unsafe void UnsafeAccess(int a, int b)
{
    fixed (int* pi = &intarray[0])
    {
        int temp = pi[a];
        pi[a] = pi[b];
        pi[b] = temp;
    }
}
static Unsafe void Main(string[] args)
{
    for (int i = 0; i < testCount; i++)
    {
        SafeAccess(0, i);
    }
    for (int i = 0; i < testCount; i++)
    {
        UnsafeAccess(0, i);
    }
}
```

Safe Loop:

I examined the code generated by the 64-bit JIT compiler for the SafeAccess loop (which was inlined into Main by the JIT). Vance Morrison posted a useful article describing how to

accomplish this from within Visual Studio:
http://blogs.msdn.com/vancem/archive/2006/02/20/535807.aspx

```
00000642`801501f0 418b08          mov     ecx,dword ptr [r8]
00000642`80150
00000642`80150
00000642`80150
00000642`80150
00000642`80150
00000642`80150
```


There are 7 instru
loop body after op

Unsafe Loop:

By contrast, the unsafe version is a mess. UnsafeAccess is larger MSIL (50 bytes vs 31) because Unsaf accesses require more MSIL instructions than safe ones. Given an array and index on the evaluation st array accesses require only a single 1-byte instruction: ldelem. The C# compiler generates a much mo complex sequence for Unsafe accesses:

```
IL_000c:  /* 06   |                */ ldloc.0 // &array[0]
IL_000d:  /* D3   |                */ conv.i
IL_000e:  /* 02   |                */ ldarg.0 // index
IL_000f:  /* D3   |                */ conv.i
IL_0010:  /* 1A   |                */ ldc.i4.4
IL_0011:  /* 5A   |                */ mul
IL_0012:  /* 58   |                */ add
IL_0013:  /* 4A   |                */ ldind.i4
```

Ignoring the first and third instructions, which are used to get the array and index, there are six instruc bytes) required to load an array element. These extra instructions make UnsafeAccess larger than Safe When determining which methods should be inlined by the JIT, one of the most highly weighted factors of the inlinee method. In this case UnsafeAccess was rejected for inlining, and because of this, the ran at &intarray[0] could not be removed. In fact the unsafe loop variant actually caused more runtime ran to occur than the safe variant!

Finally, the presence of a pinned variable inhibits many optimizations in the 64-bit JIT. As a result, the g code for UnsafeAccess is far worse than that of the safe variant. Keep in mind that the following excer only the UnsafeAccess method itself, and does not even include the the loop in Main, as the SafeAccess above does.

```
image00000000_00e40000!Arrays.UnsafeAccess(Int32, Int32):
00000642`80150260 4883ec38        sub     rsp,38h
00000642`80150264 448bc1          mov     r8d,ecx
00000642`80150267 48c744242000000000 mov   qword ptr [rsp+20h],0
00000642`80150270 48b9102e352000000000 mov rcx,20352E10h
00000642`8015027a 488b09          mov     rcx,qword ptr [rcx]
00000642`8015027d 488b4108        mov     rax,qword ptr [rcx+8]
00000642`80150281 4885c0          test    rax,rax
00000642`80150284 7641            jbe     00000642`801502c7
00000642`80150286 488d4110        lea     rax,[rcx+10h]
00000642`8015028a 4889442420      mov     qword ptr [rsp+20h],rax
00000642`8015028f 4d63c8          movsxd  r9,r8d
00000642`80150292 488b442420      mov     rax,qword ptr [rsp+20h]
00000642`80150297 468b0488        mov     r8d,dword ptr [rax+r9*4]
00000642`8015029b 4863d2          movsxd  rdx,edx
00000642`8015029e 488b442420      mov     rax,qword ptr [rsp+20h]
00000642`801502a3 8b0c90          mov     ecx,dword ptr [rax+rdx*4]
00000642`801502a6 488b442420      mov     rax,qword ptr [rsp+20h]
00000642`801502ab 42890c88        mov     dword ptr [rax+r9*4],ecx
00000642`801502af 488b442420      mov     rax,qword ptr [rsp+20h]
```

```
00000642`801502b4 44890490        mov     dword ptr [rax+rdx*4],r8d
00000642`801502b8 48c744242000000000 mov   qword ptr [rsp+20h],0
00000642`801502c1 4883c438        add     rsp,38h
00000642`801502c5 f3c3            rep ret
```


Conclusio

Unsafe array ac
pinning, and as
understand that
'safety tax' it is
viewing disasse

-Matt Grice

What's in NetFX 3.5 SP1?

August 15, 2008, 5:19 pm

≫ Next: Improvements to NGen in .NET Framework 4
≪ Previous: Performance implications of unmanaged array accesses

Long time, no blog.

Since the NetFX 3.5 Service Pack is available, now, I figured I'd put up a quick rundown of what we (the CLR CodeGen team) contributed to the package. I'm not going into nitty-gritty details, but just to give you an idea of what's in it, and perhaps inspire you to go install it. A quick note: Unless otherwise noted, all changes impact both x86 & x64.

NGen infrastructure rewrite: the new infrastructure uses less memory, produces less fragmented NGen images with much better locality, and does so in dramatically less time. What this means to you: Installing or servicing an NGen image is much faster, and cold startup time of your NGen'ed code is better.

Framework Startup Performance Improvements: The framework is now better optimized for startup. We've tweaked the framework to consider more scenarios for startup, and now layout both code & data in the framework's NGen images more optimally. What this means to you: Even your JIT code starts faster!

Better OS citizenship: We've modified NGen to produce images that are ASLR capable, in an effort to decrease potential security attack surface area. We've also started generating stacks that are always walkable using EBP-chaining for x86. What this means to you: Stack traces are more consistent, and NGen images aren't as easily used to attack the system.

Better 32-bit code quality: The x86 JIT has dramatically improved inlining heuristics that result in generally better code quality, and, in particular, much lower "cost of abstraction". If you want to author a data type that only manipulates a single integer, you can wrap the thing in a struct, and expect similar performance to code that explicitly uses an integer. There have also been some improvements to the 'assertion propagation' portion of the JIT, which means better null/range check elimination, as well as better constant propagation, and slight better 'smarts' in the JIT optimizer, overall. What this means to you: Your managed code should run slightly faster (and sometimes dramatically faster!). Note to 64 bit junkies: We're working on getting x64 there, too. The work just wasn't quite there in time.

Anyway, go forth & download!

-Kev

Improvements to NGen in .NET Framework 4

May 3, 2009, 3:41 pm

≫ Next: JIT ETW tracing in .NET Framework 4
≪ Previous: What's in NetFX 3.5 SP1?

.NET Framework 4 is our first release since we shipped FX 3.5 SP1 (FX 4 beta 1 is now available here: http://msdn.microsoft.com/en-us/vstudio/dd582936.aspx). FX 3.5 SP1 contained major changes to NGen – features that improved startup performance, security, NGen time and compilation working set – described at length in this MSDN article: http://msdn.microsoft.com/en-us/magazine/dd569747.aspx.

In FX 4 we shifted our primary focus from startup performance to framework deployment. As many of you are aware, the performance win from pre-compiling your application using NGen comes at a cost – the time taken to generate the NGen images on the end-user machine. In .NET 4 we've lowered that cost substantially in two ways. First, we've made NGEN multiproc-aware. In many cases, your assemblies (and ours) will now NGEN about twice as fast! In addition, we've substantially reduced the number of situations in which we need to regenerate NGen images. Our new Targeted Patching feature means that for many .NET Framework patches, we can now modify just the affected assemblies and not

have to re-NGEN any other dependent assemblies. Combined, these two features mean you and your use
spend a lot less time running NGEN. Best of all, you don't have to do anything to get these benefits – the
automatically when you use .NET 4.

We thought you may want to learn a little more about how these features work and the situations in whic
get these benefits. In addition, since .NET 4 is a SxS release, there is a bit of complexity under the covers
sure assemblies get NGen-ed against the matching runtime. In general things will just work the way you
guess or expect them to, but below, I'll go into some more detail about exactly what happens in various ir
cases.

If you have any comments or questions about any of these, we'd love to hear from you. Most of these feat
available in the FX

- **NGen SxS**
- **Multi-pro**
- **Targeted**
  managed
- **No NGen**
  applicatio

NGen SxS

To a large extent
We've now archit
and 4.0 MSIL asse
application runtim
against. Thus,

%WINDIR%\Micr
image compiled a

%WINDIR%\Microsoft.NET\Framework\v4.0.xxxxx\ngen.exe install <2.0 assembly> will generate an N
image compiled against the 2.0 runtime (if .NET Framework 2.0/3.0/3.5 is installed on the machine. Note
applications built against .NET Framework 2.0 won't run against .NET Framework 4.0 by default, and thus
default we don't NGen 2.0 assemblies unless CLR 2 is installed).

%WINDIR%\Microsoft.NET\Framework\v4.0.xxxxx\ngen.exe install <2.0 assembly> /ExeConfig:<Path
EXE> will generate an NGen image compiled against the 4.0 runtime.

%WINDIR%\Microsoft.NET\Framework\v4.0.xxxxx\ngen.exe install <2.0 EXE with a config file that in
4.0 as the preferred runtime> will generate an NGen image compiled against the 4.0 runtime.

%WINDIR%\Microsoft.NET\Framework\v4.0.xxxxx\ngen.exe install <2.0 assembly> /ExeConfig:<Path
EXE with a config file that indicates 4.0 as the preferred runtime> will generate an NGen image compil
the 4.0 runtime.

There was also work required to make the 2.0 NGen Service (clr_optimization_v2.0.50727_32|64) work S
the 4.0 NGen Service (clr_optimization_v4.0.xxxxx_32|64). Only one service (the latest) is active at any ti
installing FX 4 disables the 2.0 service and it is re-enabled if/when FX 4 is uninstalled.

Multiproc NGen

NGen is now aware of multiple processors/cores and can compile up to 6 assemblies in parallel (the parall
the level of assemblies). To avoid impacting foreground activities, NGen only runs in this aggressive mode
assemblies are being compiled synchronously i.e. the NGen Service still runs on one processor. Synchronou
commands (such as ngen.exe install <assembly>, ngen.exe update, ngen.exe ExecuteQueuedItems) will n
multiple processors/cores whenever possible (we also factor in amount of RAM when determining how ma
cores/processors to use). Since the parallelism is at the level of assemblies, the most effective way to enab
multiple processors for NGen is to do the following:

ngen.exe install /queue:1 <MyImportantAssembly#1>

ngen.exe install /queue:1 <MyImportantAssembly#2>

…

ngen.exe install /queue:1 <MyImportantAssembly#N>

ngen.exe install /queue:3 <MyAssembly#N+1>

….

ngen.exe install /queue:3 <MyAssembly#M>

ngen.exe ExecuteQueuedItems 1 //Synchronously compiles all important (priority 1) assemblies during se
multiple cores whenever possible; other (priority 3) assemblies will be compiled in the background by the
Service at machine idle-time.

As part of this work we reworked FX 4 set up to use the NGen pattern above.

Targeted Patching

NGen images thus far have been nothing more than "cached JIT-compiled code and CLR data structures" -
consequence they're completely fragile; any change to the underlying CLR or to any managed dependenc
invalidates them and requires them to be regenerated. For example, any change to the CLR or to basic as
like mscorlib and System that all/most assemblies depend upon invalidates all NGen images installed on th
machine. Since a machine could have several hundreds of NGen images, the cost of regenerating them af
servicing events is high. In CLR 4 we took a first step towards making NGen images less fragile to avoid the
cost associated with .NET Framework updates. In particular, FX updates that only involve fixes to bodies o
methods (that aren't generic, and aren't inlined across NGen image boundaries) will no longer require re
dependent NGen images (the old NGen images can be used with the new serviced dependency). Althoug
sound trivial (and beg the question why the system didn't have this attribute to begin with ☺), since NGe
had never been architected to be anything but serialized JIT-ed code and CLR data, accomplishing this tur
be major feat. From reworking hardbinding, doing major performance work to recover the perf impact, t
an IL post-processing tool that normalizes metadata tokens, detects Targeted Patching-compatible vs. inc
changes, and flags compatible changes such that existing NGen images can rewired to the updated deper
plugging that tool into our build system, and revising NGen cross-module inlining rules, this is a major effo
several of us have been working on for several months.

Some of the work for this (such as the changes to hardbinding and corresponding the performance work) a
included in the beta 1 build, but this feature will really come online once we start servicing FX 4. You can fi
more in the Channel 9 video on Targeted Patching.

NGen in partial trust

In FX 2 NGen images can be generated by running commands such as **ngen.exe install <Path to assembly**
intranet share> and the generated image loaded in applications running in partial trust. We believe NGe
aren't used in partial trust applications very much and our current model was broken, so we've disabled lo
NGen images in partial trust in FX 4. In the future (post CLR 4) we intend to rework this as part of simplifyi
overall NGen story (i.e. change the model where the only way to generate NGen images is to issue comm
an elevated command prompt). If you're using NGen in a partial trust application today, we'd like to hear

Surupa Biswas

CLR Codegen Team

JIT ETW tracing in .NET Framework 4

May 11, 2009, 1:56 am

>> Next: Tail Call Improvements in .NET Framework 4

<< Pre

If you
the co
the co
(sort o
decide
in som
the JIT
inform

First a
in gen
perfor

For Windows XP, Widows Server 2003, Windows Vista and Windows 7, you already have everything you need on your machine. For Windows 2000, you'll need to download the *Microsoft Windows 2000 Server Resource Kit*. According to the gurus around here previous to Windows Vista registering an ETW provider was not cheap, so the runtime only does it when requested to. On Vista and newer OSes, it is cheap enough to do all the time, so you only need to request it on pre-Vista OSes.You request it by setting the following environment variable before running the application you are interested in:

```
SET COMPLUS_ETWENABLED=1
```

To start logging ETW events do this:

```
logman start clrevents -p {e13c0d23-ccbc-4e12-931b-d9cc2eee27e4} 0x1000 5 -ets
```

There are lots more options to tweak here, but the important part is the GUID (the CLR ETW provider GUID), the mask 0x1000 (the JitTracingKeyword), and the level 5 (everything). More information about logman.exe can be found at http://technet.microsoft.com/en-us/library/bb490956.aspx. After you've started ETW, run your scenario, and then stop ETW as follows:

```
logman stop clrevents -ets
```

This will create clrevents.etl.

[Begin Edit 1/27/2010]
Several people have reported that on Vista and Win7 (and their 2008 server counter parts), one additional step is needed so that the events can be properly decoded. It is my understanding this only needs to be done once, but I am no expert here on ETW. It is also my understanding that this needs to be run from an elevated command prompt. I am merely placing this here to help others. Also you will need to replace the path with the equivalent path on your machine/setup/configuration.

```
wevtutil im c:\windows\microsoft.net\framework\v4.0.21006\clr-etw.man
```

[End Edit 1/27/2010]

To decode it further run this:

```
tracerpt clrevents.etl
```

This will create 2 files: dumpfile.csv and summary.txt. The former has all the events, the latter gives a nice summary of the events. On Vista, tracerpt will generate dumpfile.xml instead of dumpfile.csv. I ran this on a Vista machine so I'm going to deal with the XML format, but the csv format is similar.

Here is a sample MethodJitInliningSucceeded event:

```
<Eventxmlns="http://schemas.microsoft.com/win/2004/08/events/event">
    <System>
        <ProviderName="Microsoft-Windows-DotNETRuntime"Guid="{e13c0d23-
ccbc-4e12-931b-d9cc2eee27e4}" />
        <EventID>185</EventID>
        <Version>0</Version>
        <Level>5</Level>
        <Task>9</Task>
        <Opcode>83</Opcode>
        <Keywords>0x1000</Keywords>
        <TimeCreatedSystemTime="2009-04-14T14:31:52.168851900Z" />
        <CorrelationActivityID="{00000000-0000-0000-0000-000000000000}" />

<ExecutionProcessID="15476"ThreadID="16936"ProcessorID="3"KernelTime="90"UserT
/>
        <Channel />
        <Computer />
    </System>
    <UserData>
```

```
<MethodJitInliningSucceededxmlns='myNs'>

<MethodBeingCompiledNamespace>Factorial</MethodBeingCompiledNamespace>

<MethodBeingCompiledName>Main</MethodBeingCompiledName>

<MethodBeingCompiledNameSignature>void  (class System.String[])
</MethodBeingCompiledNameSignature>

</InlineeNa

<

</UserD

<RenderingInfoCulture="en-US">

<Level>Verbose </Level>

<Opcode>JitInliningSucceeded </Opcode>

<Keywords>

<Keyword>JitTracingKeyword </Keyword>

</Keywords>

<Task>CLR Method </Task>

<Message>MethodBeingCompiledNamespace=Factorial;
MethodBeingCompiledName=Main;
MethodBeingCompiledNameSignature=void  (class System.String[]);
InlinerNamespace=Factorial;
InlinerName=Main;
InlinerNameSignature=void  (class System.String[]);
InlineeNamespace=Factorial;
InlineeName=fact;
InlineeNameSignature=unsigned int32  (unsigned int32);
ClrInstanceID=13 </Message>

</RenderingInfo>

</Event>
```


We'll focus in on the UserData element because it looks the prettiest. First you have 3 sets triples: MethodBeingCompiled, Inliner, and Inlinee crossed with Namespace, Name, and Sig I think the namespace, name and signature are fairly obvious, but the reason they are sepa instead of pretty printed into a simple element is a matter of performance. ETW is suppose lightweight. Computing these strings is **not** lightweight, but we felt we made a nice trade- enough information to be useful, but don't spend time on making it prettier than it needs to MethodBeingCompiled is the method the VM asked the JIT to generate code for. Inliner refe the method that the JIT is trying to generate code for. If the Inliner is not the same as the MethodBeingCompiled it is because the JIT decided to **try** and inline the code for Inliner int code for MethodBeingCompiled rather than generate a call to Inliner. However, just becaus see a method showing up as the Inliner, it doesn't mean the JIT has actually succeeded in i it. Finally the Inlinee refers to the method that the JIT is trying to inline (rather than genera to).

If this was a MethodJitInliningFailed event it would have 2 additional elements: FailAlways FailReason. If FailAlways is true, it is a hint back to the JIT and VM that inlining will always the given Inlinee, regardless of the situation, and so subsequent compiles will be able to al inlining attempt faster (this is what FailReason of "It is marked as 'INLINE_NEVER'" means) FailReason is a free-form text field. Originally this came from some internal code we had t in our debugging. The text is often cryptic, and is not localized. Part of the reason we did n localize it was performance. The other part is that these strings are often so cryptic many E speakers will have a hard time with them. For us we often end up treating them like a GUID search the sources for the places where that reason is returned to figure out why a particul failed. So if English is not your native language, don't fret too much, you're probably not m much. That covers the 2 most common events: MethodJitInliningSucceeded and MethodJitInliningFailed.

Now we move onto the only other 2 events for this keyword: MethodJitTailCallSucceeded ar MethodJitTailCallFailed. The first 9 elements are the same (except replace Inliner with Call Inlinee with Callee). The next element is the TailPrefix element. This element tells whethe tail. IL prefix was present. For x86 this currently will always be true because the x86 JIT do generate tail calls as an optimization. See other blog posts for why tail calls are a good optimization that the x64 and IA64 JIT often performs even without the tail. il prefix. We fin useful because, although some compilers emit the tail. prefix in IL, they do not have any so syntax to expose it, so the programmer does not know looking only at the source what was

For MethodJitTailCallSucceeded, you get a TailCallType. This is an enum, which has names t should be localized down in the Message element. Due to a bug in Beta 1 you will only get numeric values. Also the pre-Vista versions of tracerpt do not do the translation from value

string. So on post Beta 1 builds with Vista or newer, you should the pretty names, otherwis
the numbers. I will list both, especially because the number still appear in the UserData el
OptimizedTailCall, 0, means a typical tail call, where the outgoing arguments are pushed in
incoming parameter slots, the epilog is executed but instead of returning, it ends with a jur
new method. RecursiveLoop, 1, means a recursive tail call, where the JIT sees that the met
calls itself, and replaces the call with a jump back to the start of the method (thus skipping
prolog and epil
directly and mu
normal calls an
prevent stack o
TailPrefix=false

For MethodJitTa
similar to Meth
we hope to exp

So for the few p
you can now se
enable you to re
remember thou
methods was or wasn't inlined in one version doesn't mean the same will be true on a diffe
version. If a method absolutely cannot be inlined, use MethodImplOptions.NoInlining. If it
absolutely must be inlined then inline it manually (i.e. copy and paste the code). That
recommendation should never change. If however you're looking for some simple performa
boosts, you can try these events and possibly learn something. One example we found inte
that moving a try/catch or try/finally up or down the call chain can often have a big perform
impact because it might enable/disable inlining inside a very important loop.

Grant Richins
CLR Codegen Team
Update 10/8/2009 - replace em-dash in logman command line with a normal dash that you car
into a command window and it will work.

Tail Call Improvements in .NET Framework 4

May 11, 2009, 4:50 am

» Next: Array Bounds Check Elimination in the CLR
« Previous: JIT ETW tracing in .NET Framework 4

First a little background reading before going into tail call improvements in CLR 4 - David Broman did an excellent job at covering the basics in his post here: http://blogs.msdn.com/davbr/archive/2007/06/20/enter-leave-tailcall-hooks-part-2-tall-tales-of-tail-calls.aspx. He also captured a mostly complete list of the restrictions as they stood for CLR 2 here: http://blogs.msdn.com/davbr/pages/tail-call-jit-conditions.aspx.

The primary reason for a tail call as an optimization is to improve data locality, memory usage, and cache usage. By doing a tail call the callee will use the same stack space as the caller. This reduces memory pressure. It marginally improves the cache because the same memory is reused for subsequent callers and thus can stay in the cache, rather than evicting some older cache line to make room for a new cache line.

The other usage of tail calls are in recursive algorithms. We all know from our computer science theory classes that any recursive algorithm can be made into an iterative one. However, just because you can doesn't mean you always want to. If the algorithm is naturally tail recursive, why not let the compiler do the work for you? By using a tail call, the compiler effectively turns a tail recursive algorithm into a loop. This is such an important concept that the JIT has a special path just for turning functions that call themselves in a tail recursive manner into loops. This is a nice performance win because beyond the memory improvements mentioned previously, you also save several instructions by not executing the prolog or epilog multiple times. The compiler is also able to treat it like a loop and hoist out loop invariants so they are also only executed once instead of being executed on every iteration.

In CLR 2 the 64-bit JIT focused solely on the 'easy' cases. That meant that it generated code that did a tail call whenever it could because of the memory benefits. However, it also meant that sometimes when the IL requested a tail call using the "tail." instruction prefix, the JIT would not generate a tail call because it wasn't easy. The 32-bit JIT had a general purpose tail call mechanism that always worked, but wasn't performant enough to consider it an optimization, so it was only used when the IL requested a tail call.

For CLR 4 the fact that the x64 JIT would sometimes not honor the "tail." prefix prevented functional languages like F# from being viable. So we worked to improve the x64 JIT so that it could honor the "tail." prefix all of the time and help make F# a success. Except where explicitly called out, x86 and IA64 remain unchanged between CLR 2 and CLR 4, and the remainder of this document refers solely to the x64 JIT. Also this is specific to CLR 4, all other versions of the runtime or JIT (including service packs, QFEs, etc.) might change this in any number of ways.

The 64-bit calling convention is basically a caller-pops convention. Thus the 'easy' cases basically boiled down to where the JIT could generate code that stored the outgoing tail call arguments in the caller's incoming argument slots, execute the epilog, and then jump to the target rather than returning. The improvements for CLR 4 came in two categories: Fewer restrictions on the optimized/easy cases and a solution for the non-easy cases. The end result is that on x64 you should see shorter call stacks in optimized code because the JIT generated more tail calls (don't worry this optimization is turned off for debug code), and functional programmers (or any other compiler or language that relies on tail calls and uses the "tail." prefix) no longer need to worry about stack overflows. The down side of

course is that if you have to debug or profile optimized code, be prepared to deal with call s
that look like they're missing a few frames.

Reducing restrictions on the easy cases

The more often we can tail call, the more likely programs will benefit from the optimization
the second link above you can see that there were a lot of restrictions on when the JIT coul
perform a tail call. We looked at each one of these and tried to see if we could reduce or e
them.

- The call/call

We reduced this
method that ret
callee was obvi
return value is j
the tail call is le
with the tail. pr
generator shou
- The caller or

We changed the
that for value ty
should pass a 'h
buffer to the ca
buffer. The CLR
return buffer int
code has at least one less copy, and it enables one more case to be 'easy'.

- The caller is a shared generic method.

The above restriction is now removed!

- The caller is varargs
- The callee is varargs.

These have been partially removed. Specifically, we treat the caller as only having its fixed
arguments and the callee as having all of its arguments. Then we apply all of the other rul
also worth mentioning that the CLR deviates from the native X64 calling convention by add
another 'hidden' argument called the vararg cookie. It stores the GC properties of the actu
(important for the variadic part). This hidden argument is also counted as part of the signa
the purposes of the other rules.

- The runtime forbids the JIT to tail call. (*There are various reasons the runtime may disa
calling, such as caller / callee being in different assemblies, the call going to the applica
entrypoint, any conflicts with usage of security features, and other esoteric cases.*)

This part applies to all platforms. We were able to reduce the number of cases where the r
refused a tail call due to security constraints. Mainly the runtime stopped caring so much a
exact assembly and instead looked at the security properties. If performing the call as a ta
would not cause an important stack frame to disappear from the call stack, it is now allowe
Previous to this change tail calls across modules or to unknown destinations (due to calli o
were rare, now they are significantly more common.

- The callee is invoked via stub dispatch (*i.e., via intermediate code that's generated at ru
optimize certain types of calls*).

The above restriction is now removed!

- For x64 we have these additional restrictions:
  - …
  - For all of the parameters passed on the stack the GC-ness must match between the c
  callee. (*"GC-ness" means the state of being a pointer to the beginning of an object m
  by the GC, or a pointer to the interior of an object managed by the GC (e.g., a byref fi
  neither (e.g., an integer or struct).*)

The above x64 restriction is now removed!

A solution for the non-easy cases

Previously even if the IL had the "tail." prefix, if the call did not match the requirements to l
the optimized easy tail calls cases, it was turned into a regular call. This is disastrous for r
algorithms. In many cases this prevented F# programs from running to completion without
stack overflows. So for CLR 4 we added an alternate code path that allowed tail call like pr
for these specific scenarios, where it was needed, but it was not easy.

The biggest problem to overcome is the x64 calling convention. Because of the calling con
if a tail call callee needs more argument space than the caller has, the code is wedged bet
rock and a hard place. The solution involved a little stack trickery. The JIT generates a mos
normal call instead of a tail call. However instead of calling the target directly it calls throu
special helper which works some magic on the stack. It logically unwinds the stack to be a
caller has returned. Then it adds a fake method to the stack as if it had been called, and th
jumps to the callee. This fake method is called the TailCallHelper. It takes no arguments, a
as unwind data and the calling convention are concerned calls the callee with no outgoing
arguments, but has a dynamically resizing locals area (think _alloca for C/C++ programmers
stackalloc for C# programmers). Thus this method is still caller-pops, but the tail call helpe
dynamically change how much to pop based on a given callee.

This stack magic is not cheap. It is significantly slower than a normal call or a tail call. It a
consumes a non-trivial amount of stack space. However the stack space is 'recycled' such
you have a tail recursive algorithm, the first tail call that isn't 'easy' will erect the TailCallH
stack frame, and subsequent non-easy tail calls may need to grow that frame to accommoc
the arguments. Hopefully once the algorithm has gone through a full cycle of recursion the
TailCallHelper stack frame has grown to the maximum sized needed by all the non-easy cal
involved in the recursion, and then never grows, and thus prevents a stack overflow.

As you can see the way the non-easy case is handled is also not fast. Because of that, the
uses the non-easy helper unless the IL requires it with the "tail." prefix. This was also the
force behind trying to reduce the number of restrictions on the 'easy' cases.

Grant Richins
CLR Codegen Team

[Updated 6/25/2009 to make a few minor clarifications based on internal feedback]

Array Bounds Check Elimination in the CLR

August 13, 2009, 4:46 pm

>> Next: JIT ETW Inlining Event Fail Reasons
<< Pre

Intr

One a
"mana
array
say th
an app
is safe

We're
resear
implemented some subset of these techniques. Putting "array bounds check elimination" into bing.com yielded a large number of relevant papers, many of which I've read and enjoyed; I'd imagine a competitor's search site would do the same ☺.

This blog post will explore what the CLR's just-in-time compilers do and do not do in this area. I'll of course highlight the good cases, but I'm also going to be brutally honest, and expose many examples where we could potentially eliminate a range check, but don't. The reader (and, for that matter, the author, who didn't implement this stuff himself) should keep in mind an important constraint: these are, in fact dynamic JIT compilers, so any extra optimization that slows the compiler down must be balanced against the gains of that optimization. Even when we run the JIT "offline", via the NGEN tool, users are sensitive to compiler throughput. So there are many things we *could* do, but all take CLR developer effort, and some of them use up our precious compilation time budget. That excuse being made, it's up to us to be clever and figure out how to do some of these optimizations efficiently in the compiler, and we'll certainly try to do more of that in the future.

The JIT compilers for x86 and x64 are currently quite different code bases, and I'll describe the behavior of each here. The reader should note however, that we intend to unify them at some point in the not-too-distant future. The x86 JIT is faster in terms of compilation speed; the x64 JIT is slower, but does more interesting optimizations. Our plan is to extend the x86 codebase to generate x64 code, and incorporate some of the x64 JIT's optimizations without unduly increasing compilation time. In any case, performance characteristics of JITted code on x64 platforms is likely to change significantly when this unification is achieved.

When I show examples where we don't eliminate bounds checks, I will when possible give advice that will help you stay within boundary of idioms for which we can. I'll discuss things we might be able to do in the future, but I'm not in a position to give any scheduling commitments on when these might be done. I can say that any reader feedback on prioritization will be taken into account.

Code Gen for Range Checks

Before we start considering when we eliminate range checks, let's see what the code generated for a range check looks like. Here is bounds-check code generated by the CLR's x86 JIT for an example array index expression a[i]:

```
IN0001: 000003      cmp     EDX, dword ptr [ECX+4]          // a in ECX, i
in EDX
IN0002: 000006      jae     SHORT G_M60672_IG03             // unsigned
comparison
```

In the first instruction, EDX contains the array index, and ECX + 4 is the address of the length field of the array. We compare these, and jump if the index is greater than or equal to the length. The jump target, not shown, raises a `System.IndexOutOfRangeException`. A sharp-eyed reader might wonder: the semantics require not only that the index value is less than the array length, but also that it is at least zero. Isn't that two checks? How did they get away with only one comparison and branch? The answer is that we (like many other systems) take advantage of the wonders of unsigned arithmetic – the x86 "jae" instruction interprets its arguments as unsigned integers (it's the unsigned equivalent of "jge", if that's more familiar to some readers). The type of the length of an array, and an expression used to index into an array, is Int32, not UInt32. So the maximum value for either of these is 2^{31}-1. Further, we know that the array length will be non-negative. So if we convert the array length to a UInt32, it doesn't affect its value. The index value, however, *might* be negative. If it is, casting its bit pattern to UInt32 yields a value that is *at least* 2^{31}. So both cases, when the index value is negative, or when it is larger than the array length, are handled by the same test.

In an NGEN image, we try to separate out code we expect to never be executed (code that is so "cold" that it's at absolute zero!), hoping to increase working set density, especially during startup. We expect bounds-check failures to be in this category, so we put the basic blocks for failure cases on cold pages.

Bounds-check removal cases

Now we'll examine some test cases, starting with some simple ones.

Simple cases

The good news is that we *do* eliminate bounds checks for what we believe to be the most c
form of array accesses: accesses that occur within a loop over all the indices of the loop. S
is no dynamic range check for the array access in this program:

```
static void Test_SimpleAscend(int[] a) {

    for (int i = 0; i < a.Length; i++)

        a[i] = i;  // We get this.

}
```


Unfortunately, v

```
static vo

    for

        a

}
```

Some older prog
DEC PDP-8, if m
not auto-increment. They may have passed this habit down to middle-aged programmers (
I am one), and so on. There's also a somewhat more currently-valid argument that hardwa
generally supports a comparison to zero without requiring the value zero to be placed in a r
In any case, while the JIT compiler should arguably eliminate the bounds check for the desc
form of the loop, we don't today, and the cost of the bounds check probably outweighs any
other advantages. So:

- **Advice 1:** if you have the choice between using an ascending or descending loop to
  an array, choose ascending.

I've put the array access on the left-hand side of an assignment in both these examples, bu
works independently of the context in which the array index expression appears (as long as
within the loop, of course).

Do we track equalities with the length of a newly allocated array?

Here is a case in which the x86 JIT does not eliminate the bounds check:

```
static int[] Test_ArrayCopy1(int n) {

    int[] ia = new int[n];

    for (int i = 0; i < n; i++)

        ia[i] = i;  // We do not get this one.

    return ia;

}
```

No excuses here: there's no reason not to get this, the JIT compiler ought to know that `n` is
length of the newly allocated array in `ia`. In fact, the author of such code might think he wa
the JIT compiler a favor, since comparison with a local variable `n` might seem cheaper than
comparison with `ia.Length` (though this isn't really true on Intel machines). But in our syst
least today, this sort of transformation is counterproductive for the x86 JIT, since it prevents
bounds check from being eliminated. We may well extend our compiler(s) to track this sort
equivalence in the future. For now, though, you should follow this piece of practical advice

- **Advice 2:** When possible, use "a.Length" to bound a loop whose index variable is u
  index into "a".

The x64 JIT *does* eliminate the range checks here, by hoisting a test outside the loop, comp
with `ia.Length`. If this check fails, it throws an `IndexOutOfRangeException`. This is somewha
problematic, since without this optimization the program would execute `ia.Length` iteration
loop before throwing an exception, and strict language semantics would require those to b
executed if they could possibly have a side-effect visible outside the method (which this ex
does not in fact have – though proving it requires your compiler to do enough escape analy
know that the allocated array that is written to has not leaked outside the method). This se
ambiguity is the subject of some internal debate, and we'll eventually reach consensus on
how/when to incorporate such tests in a unified JIT,or whether we need to ensure strict sen
perhaps by generating multiple copies of loop bodies, as we'll discuss below. (It's interesti
note that the hoisted test and throw would be justified by assuming the
`CompilationRelaxationsAttribute` defined in section I.12.6.4 of the ECMA CLI specification
bounds-check error exceptions everywhere – whereas the specification requires it to be giv
explicitly.) In any case, we should emphasize that, as far as we know, this is a "theoretical"
concern only – we don't know of any actual customer code whose correctness is affected b
issue.

Redundant array accesses

OK, while we're slightly embarrassed by the previous "multiple names for the length" case,
cheer ourselves up with something we do well. We're pretty good at eliminating redundant
checks. In the method:

```
static void Test_SimpleRedundant(int[] a, int i) {

    k = a[i];

    k = k + a[i];

}
```

bounds-check code is generated for the first instance of "a[i]", but not the second. In fact,
JIT treats it as a common subexpression, and the first result is re-used. And this works not

within “basic blocks” – it can work across control flow, as demonstrated by:

```
static void Test_RedundantExBB(int[] a, int i, bool b) {
    k = a[i];
    if (b) {
        }
    } els
        }
    }
}
```


As before, the f
The x86 JIT als
first read of “a[

It is not the cas
common subexp

```
static void Test_RedundantNotCse(int[] a, int i, int j) {
    k = a[i];
    a[j] = i;
    k = k + a[i];
}
```

The JIT compiler obviously can’t tell whether “i” and “j” will have the same value at runtim
can tell that they *might,* and that if they do, the “a[i]” on the last line will return the value
there on the second line. So we cannot treat the “a[i]” expressions on the first and last line
body as common subexpressions. But the assignment on the second line can’t affect the *le*
the array “a,” so in fact the bounds check for the first line “covers” the “a[i]” on the third li
generated code accesses the array without a bounds check (in both JITs).

Arrays as IEnumerable

Arrays implement the IEnumerable interface, which raises a reasonable question: if you en
over the elements of an array using C#’s `foreach` construct, do you get bounds checks? For
example:

```
static int Test_Foreach(int[] ia) {
    int sum = 0;
    foreach (int i in ia) {
        sum += i;
    }
    return sum;
}
```

Happily, we do eliminate the bounds checks in this case. However, there is a little quirk he
cases listed, this one is the only one that is sensitive to whether the original source progra
this case) was compiled to IL with the /optimize flag. The default for csc, the C# compiler,
optimize, and in this mode it produces somewhat more verbose IL for the range check that
fit the pattern that the JIT compiler looks for. So:

- **Advice 3:** if you’re worried about performance, and your compiler has an optimizati
uh, use it!

Arrays in global locations; concurrency

Here’s a case where we don’t eliminate the bounds check, but where we aren’t too embarra
this failure:

```
static int[] v;
…
static void Test_ArrayInGlobal() {
    for (int i = 0; i < v.Length; i++)
        v[i] = i;
}
```

At first glance, this seems exactly the same as our first, simplest example, Test_SimpleAsce
difference is that Test_SimpleAscend took an array argument, whereas Test_ArrayInGlobal’s
accessed via a static variable, accessible to other threads. This makes static elimination o
bounds check for “v[i]” at the very least dicey. Let’s say we did, and that “v” initially holds
of length 100. On the iteration when “i” reaches (say) 80, we check “i < v.Length”, and it’s
true. Now another thread sets “v” to an array whose length is only 50. If we go ahead wit
array store without a dynamic check, we’re writing off the end of the array – type-safety an
security are lost, game over. (Obviously, the same reasoning would apply for an array held
location accessible to multiple threads – an object field, element of another array, anything
local to the running thread.)

So we don't do this, for good and solid reasons. If we cared enough, there *is* a technique th
allow us to eliminate these bounds checks. But it would require us to couple otherwise-uni
optimizations. As it happens, the code for accessing a static variable in the presence of ap
domains can be moderately costly, so it's good to treat those as common-subexpression
candidates, and the x86 JIT does in this case (the x64 JIT does not). So the optimizer in ess
synthesizes a local variable to hold the array. If we do this, then we *are* back in the
Test_SimpleAscend situation, and the bounds-check elimination is legal. But doing the bou
check elimination *requires* that the static variable be read once into a local. So it's at least
complicated.

Parallel array

Next we consid
their structure,

```
static in
    // Th
    int s
    for
        a
        s
    }
    return sum;
}
```

Much as with Test_ArrayCopy1, the x64 JIT hoists a test comparing `ia2.Length` and `ia1.Len` immediately throwing the bounds-check exception if the test fails. If the test succeeds, rar checks for both array accesses in the loop are eliminated. The same comments about the s issues with such a test apply. The x86 JIT takes a more "purist" approach: it does not hoist so it only eliminates the bounds check for the access to the array `ia1` whose `Length` bounds index variable.

We could resolve the two approaches. The mechanisms proposed for this sort of problem i research literature have the common property that they require, at least in some cases, ge code for the loop multiple times, under different assumptions, and synthesizing some sort c determine which version of the loop should be executed – this is essentially the test that th is already creating. Generally, bounds check exceptions are rare – if the programmer wrote code above, he or she had some reason to believe that the index expression "ia2[i]" was sa we could synthesize a test on that basis. In our case above, if the compiler proved that nei argument variable "ia1" or "ia2" was modified in the loop, then a test "ia2.Length >= ia1.Le (the one the x64 JIT generates) outside the loop would allow us to execute an optimized ve the loop, with no bounds checks for either array access. If this test failed, however, we'd n execute an unoptimized version of the loop to be completely semantically correct. You'd ha evaluate this test carefully, since it's code that doesn't appear in the original program. In p in this case, you'd have to worry about whether either of "ia1" or "ia2" were null. If they a want the null pointer exception to occur at the appropriate point in execution, not in some compiler made up. So the synthesized test would have to include null tests, and take the unoptimized path if either argument is null.

As we've discussed, the x64 JIT generates the test, but not the unoptimized version of the l throws the exception "early" in that case. Under the "purist" viewpoint, this is incorrect be the test fails, the semantics require the program to execute some number of loop iteration throwing the exception, and those iterations might have side effects. In many cases, we m able to prove that the loop body does *not* have side effects, and therefore use the x64 JIT's with semantic blessing. For example, a loop that computed the sum of the loop elements i local would side-effect only that local variable – if the exception causes control flow to leav method, the value of that local becomes meaningless.

Many other patterns are amenable to this sort of synthesized test. An alternative form of Test_TwoArrays might have passed the shared length of both arrays as a separate argumer used that as the loop bound. We could do something similar, synthesizing a test of that loc vs. both array lengths.

Explicit Assertions

Another suggestion that has been made is to allow the programmer to provide the relevant the form of a contract assertion (of a flavor that would be executed in all execution modes, in a debug mode). This would essentially provide semantic "permission" to fail immediate test is violated, avoiding the need to have an unoptimized version of the loop. There are m things to be said for this sort of proposal: they can allow bounds checks to be eliminated in situations more complicated than those for which the compiler could easily infer a test, and invariants they express are often useful program documentation as well. Still, I also worry somewhat about such proposals. In many common cases, it's easy enough to infer the test should avoid *requiring* the programmer to add assertions in the easy cases. More importar programmer adds an assertion expecting it to eliminate a bounds check, how does the tool indicate whether he or she has been successful? And, if not, why not? These sorts of issue some more thought.

Still another path would be to have a custom annotation like [OptimizeForSpeedNotSpace], the programmer to tell us that the performance of this method is important enough that we apply optimizations that we wouldn't generally apply because they increase code size – *i.e.* especially aggressive inlining, loop unrolling, loop body replication/specialization for the re discussed here, or for other forms of specialization.

The right strategy in this area is obviously a little muddled. Constructive feedback is welcc

Copy loop

Here's another example, somewhat similar to the Test_TwoArrays case:

```
static int[] Test_ArrayCopy2(int[] ia1) {

    // An array copy loop operation.
```

```
        int[] res = new int[ia1.Length];
        for (int i = 0; i < res.Length; i++) {
            res[i] = ia1[i];
        }
        retu
    }
```


As you might ex
we eliminate th
access to "ia1".
equivalences of
but it's certainl
the access "ia1
with the same c

While it would
arrays, there ar
copy loop like t

- **Advice 4:** when you're copying medium-to-large arrays, use Array.Copy, rather than copy loops. First, all your range checks will be "hoisted" to a single check outside t If the arrays contain object references, you will also get efficient "hoisting" of two r expenses related to storing into arrays of object types: the per-element "store chec related to array covariance can often be eliminated by a check on the dynamic type arrays, and garbage-collection-related write barriers will be aggregated and becom more efficient. Finally, we will able to use more efficient "memcpy"-style copy loop in the coming multicore world, perhaps even employ parallelism if the arrays are bi enough!)

Multi-dimensional Arrays

The CLR, and C#, support real multi-dimensional arrays – in contrast to C++ or Java, which support only one-dimensional arrays. To get two-dimensional arrays, you have to simulate either through classes that represent the 2-d array as a large 1-d array, and do the appropri index arithmetic, or as an "array-of-arrays." In the latter case, even if they are allocated or to form a "rectangular" 2-d array, it's hard for a compiler to prove that the array *stays* recta so bounds check on accesses to the "inner" arrays are hard to prove.

With true multi-dimensional arrays, the array lengths in each dimension are immutable (jus length of a regular 1-d array is). This makes removing of bounds checks in each dimension tractable. A related advantage is that indexing calculations become easier when the array to be "rectangular." (With a good optimizer and appropriately aggressive inlining, C++ tem class-based simulations of multidimensional arrays can get similar indexing calculation co

Unfortunately, we aren't yet able to remove any range checks for accesses in multi-dimens arrays, even in simple cases like this:

```
static int Test_2D(int[,] mat) {
    int sum = 0;
    for (int i = 0; i < mat.GetLength(0); i++) {
        for (int j = 0; j < mat.GetLength(1); j++) {
            sum += mat[i, j];
        }
    }
    return sum;
}
```

The "mat.GetLength(k)" method returns the length of "mat" in the k^{th} dimension. We'll cle need to eliminate bounds checks for multi-dimensional array accesses if we want to genera reasonable code for, say, a matrix multiplication.

- **Advice 5:** Until we get this right, I would suggest that .NET users do what many C+- numerical programmers do: write a class to implement your n-dimensional array. Th be represented as a 1-dimensional array, and the relevant accessors would convert into 1 via appropriate multiplications. We almost certainly wouldn't eliminate the b check into the 1-d array, but at least we'd only do one check!

Conclusions

First, let's accentuate the positive: we do eliminate bounds checks in some very common c the costs of bounds checks usually aren't that great when we don't eliminate them. And, a mentioned at the beginning, we have to keep in mind that the compiler we're talking about dynamic JIT compiler, so we must carefully balance adding extra optimization that slows th compiler against the gains of that optimization. Still, if we don't eliminate a bounds check should have in a small, tight loop that's important to the performance of your program, I do you'll find these excuses very satisfying. I hope this blog post convinces you that we're we of the problems. The future almost certainly holds some mechanism for applying extra con effort to methods whose performance matters a lot, either by doing the extra work in some offline build-lab compilation, or by using profile-directed feedback, user annotations of hot methods, or other heuristics. When we can do extra compiler work, bounds-check eliminat be one of the problems we address.

JIT ETW Inlining Event Fail Reasons

October 21, 2009, 3:01 pm

≫ Next: NGen: Walk-through Series
≪ Previous: Array Bounds Check Elimination in the CLR

This is a follow up post for JIT ETW tracing in .NET Framework 4. These are some of the possib
These
and ar

- "Inl
cor
the
- "Inl
run
- "Pr
- "Pr
CO
ICo
- "Inl
ver
- "Inl
MethodImplOptions.NoInlining, or the VM or JIT previously determined that the method would never be a good inline candidate and marked it as such to speed subsequent JITs that might try to inline the same inlinee.
- "Inlinee requires a security object (calls Demand/Assert/Deny)" - the inlinee is marked with mdRequireSecObject. With our current implementation, such methods need their own call frame so the corresponding Assert or Deny will end at the return of the method.
- "Inlinee is MethodImpl'd by another method within the same type" - an implementation limitation such that it can't find the right method to give to the JIT to inline (but don't worry, it still finds the right one to call). See IMetadataEmit::DefineMethodImpl for how this is done at the IL level. It is my understanding that the C# language does not allow this, but VB.NET does.
- "Targeted Patching - Method lacks TargetedPatchingOptOutAttribute" - this relates to a new feature in CLR 4 where NGEN images are more version resilient. In a nutshell, we hope to be able to apply a patch or fix to mscorlib.dll and **not** have to recompile all the other native images on the machine that depend upon it. This should only apply to methods in the .NET framework class libraries because they are the only assemblies that can opt into this feature. For more information, see this previous blog post: Improvements to NGen in .NET Framework 4.
- "Inlinee has restrictions the JIT doesn't want" - although this is in the code, in our current implementation this will never happen. It indicates that the VM needs to place a restriction on the inlinee (i.e. the inlinee can only be inlined if certain conditions are met), but the JIT doesn't allow any restrictions, so the VM has to refuse the inlining.

NGen: Walk-through Series

April 27, 2010, 10:24 am

≫ Next: NGen: Getting Started with NGen in Visual Studio
≪ Previous: JIT ETW Inlining Event Fail Reasons

Now that Microsoft Visual Studio 2010 has shipped, we thought it would be a good time to publish a series of articles focused on how to use the NGen technology and how to measure performance benefits from it. This series features hands-on style content, so get your copy of Visual Studio 2010 installed and you'll be ready to follow along.

The following topics will be covered in the 4 blog posts that make up this series.

1. NGen: Getting Started with NGen in Visual Studio
2. NGen: Measuring Warm Startup Performance with Xperf
3. NGen: Measuring Working Set with VMMap
4. NGen: Creating Setup Projects

Please use the comments section under each post to reach out to us, provide feedback and ask questions.

Pracheeti Nagarkar

CLR CodeGen Team

NGen: Getting Started with NGen in Visual Studio

April 27, 2010, 10:39 am

≫ Next: NGen: Measuring Warm Startup Performance with Xperf
≪ Previous: NGen: Walk-through Series

This is article 1 of 4 in the **NGen: Walkthrough Series**.

Hey there managed code developer. So you'd like to test drive the NGen technology in the .Net Framework? This article will walk you through how to use NGen for your existing solution in Visual Studio 2010.

To familiarize yourself with the concepts around NGen (how it works and for what style of application/library it makes sense to use), I strongly encourage you to first read through this excellent MSDN CLR Inside Out article around the Performance Benefits of NGen: http://msdn.microsoft.com/en-us/magazine/cc163610.aspx. Now assuming you have some background on when to use NGen, let's get started. I'm assuming you have identified that NGen will likely benefit your application (perhaps your application has been identified as having
gets lo
impact

For th
your a
applica
below
run. N
installe

Addi

Setting

In your Visual Studio solution, under the **Project** menu, go to **Properties** and then go to the **Build Events** tab. Under the **Post-build event command line**, click on **Edit Post-Build…**

Specify the command line as **%WINDIR%\Microsoft.Net\Framework\v4.0.30319\ngen.exe install "$(TargetPath)"**

At the time of this writing, there is no Macro in Visual Studio that points to ngen.exe, so you will need to specify the path yourself. An easy way to locate the path to ngen.exe is to open up the Visual Studio Command Prompt (All Programs -> Microsoft Visual Studio 2010 -> Visual Studio Tools -> Visual Studio Command Prompt) and type "**where ngen.exe"**.

The NGen install command creates native images for the target specified along with any static dependencies that target may have.

Note that multiple post-build events can be specified in the **Edit Post Build…** window by specifying each command on a separate line.

Under the **Project** menu go to **Properties** and in the **Debug** tab, uncheck the box for **Enable the Visual Studio Hosting Process**.

Keep in mind that you need to be intentional about the bitness of NGen.exe that you use. If your development environment is on a 64 bit machine, the application in Visual Studio by default will be created as a 32 bit application, and you should use the 32 bit ngen.exe in the Post-Build Event. However, on 64 bit machines, outside the Visual Studio development environment, the application will run against the 64 bit runtime, and so you need to NGen using the 64 bit NGen.exe tool.

(On a related note, during build, if you choose to assign base addresses to the DLLs created in your Visual Studio Solution, the way to do that is under **Project** menu go to **Properties** and in the **Build** tab click on **Advanced…**

In the **Advanced Build Settings** window, under **Output**, the **DLL Base Address** field lets you specify the base address to assign to the DLL.)

Further Reading

1> How to: Specify Build Events: http://msdn.microsoft.com/en-us/library/ke5z92ks(VS.80).aspx

2> How to install an assembly into the GAC: http://support.microsoft.com/kb/815808

Ensuring the post-build event worked

Now that you've added your post-build event to run NGen.exe on the built executables, it's important to determine 2 things –

1> Whether native images were generated successfully, AND

2> Whether native images got loaded during runtime

Ensuring native images were generated successfully

After building the solution (F6), look at the Error List window to ensure there are no errors. If there are errors during build, look at the Output window to see where the error is coming from and if it is from the NGen.exe tool.

Some common reasons why NGen.exe might have failed are –

1> Visual Studio is not being run with Administrator credentials. NGen.exe is an Admin tool and the post-build step will fail with the following error if you run without the appropriate credentials –

```
Access is denied. (Exception from HRESULT: 0x80070005 (E_ACCESSDENIED))
```

```
Administrator permissions are needed to use the selected options.  Use an administrator command prompt to complete t
```

2> The post-build event command line contains the macro $(TargetPath), but this path conta
spaces in it. Changing this to be "$(TargetPath)" should fix the problem.

Ensuring native images got loaded during runtime

A native image f
special to run a
(the IL assembly
exists for this IL

The first thing to
check for is whe
image for the as

There are a cou

1> Use the SD

Launch the
to Disk. (You may also need to check **Enable custom log path** and provide a custom log pa

Run your application.

In the **Log Categories** box, select **Native Images** and hit **Refresh**.

The format of each entry in this log is such that Application column is the name of the exe
and Description column contains the name of the assembly being loaded.

If you open the log entry, at the bottom of the text will be a "WRN: No matching native ima
found." If the native image for that assembly could not be loaded.

2> Inspect the list of loaded modules in Visual Studio: useful if running within Visual Studio

The list of modules does not contain the native image files by default if your VS Solution
created for managed code. In order to work around this problem, from the **Project** menu go to
Properties and in the **Debug** tab, mark the check box for **Enable Unmanaged Code Debu**

While debugging, from the **Debug** menu go to **Windows** and **Modules**, to see a list of t
modules loaded into the process.

Further Reading

1> Assembly Binding Log Viewer (fuslogvw.exe): http://msdn.microsoft.com/en-us/library/e74a18c4.aspx

2> More tricks on how to tell if the NGen image got loaded:
http://blogs.msdn.com/jmstall/archive/2006/10/04/debugging_5F00_ngen_5F00_code.aspx

Ensuring that the JIT did not get used during runtime

You might be wondering what the difference between the previous section "Ensuring native im
loaded during runtime" and this one is. If the native image file got loaded for an assembly, the
should not fire for it, right? Well, not quite. Sometimes, even though the native image might ha
used for the bulk of the method in that assembly, the JIT does get loaded for some methods, c
of code within the assembly being executed; for example, some generic instantiations, dynami
generated code, multiple AppDomain scenarios etc. So, even though the native image file mig
been loaded for an assembly and we may have used the native code from that assembly for m
our method execution needs, for some methods, we may have to fall back to JIT. It is important
what those methods are for your application.

Moreover, if your application has several DLLs, it may not be a trivial amount of work to ensure
each DLL had a corresponding ".ni.dll" loaded for it, by inspecting either Fusion Log Viewer co
the Modules window content. Using the JitCompilationStart MDA will help in such scenarios as

As of this writing, the easiest way to determine this is by using the JitCompilationStart Manage
Debugging Assistant. The 2 MSDN articles listed under "Further Reading" below give some ex
background information on what MDAs are, how to use them and how the JitCompilationStart M
be used.

Now let's look at the practical steps for how to set up your Visual Studio configuration correctly
the JitCompilationStart MDA will fire when it needs to.

1> Disable the Visual Studio hosting process. See http://msdn.microsoft.com/en-us/library/ms185330(VS.80).aspx for instructions.

2> If your application executable name is hello.exe, add a hello.exe.mda.config file to your sc
with the contents below and in the **Properties** set **Copy to Output Directory** to **Copy Alway**

```
<mdaConfig>
    <assistants>
        <jitCompilationStart>
            <methods>
                <match name="*" />
            </methods>
        </jitCompilationStart >
    </assistants>
```

```
</mdaConfig>
```

3> Under the menu **Debug** go to **Exceptions** and set **Managed Debugging Assistants** to **Thrown.**

4> From the P
Unmanaged C

5> Finally, in c
created in Step
create a REG_S

Yes I understan
we hope to impr

Now that the Jit
ed, an unhandle

Select Break in the dialog box to see the line of disassembly for which the JIT got loaded. The Stack window will also show the unmanaged call stack at which the break point occurs. This wi a clue as to what part of your code caused the JIT to load.

Further Reading

1> Diagnosing Errors with Managed Debugging Assistants: http://msdn.microsoft.com/en-us/library/d21c150d(VS.80).aspx

2> JitCompilationStart: http://msdn.microsoft.com/en-us/library/fw872k46(VS.80).aspx

3> Description for how JIT can get loaded for multiple AppDomain scenarios: http://msdn.microsoft.com/en-us/magazine/cc163610.aspx#S6

Wrapping it up!

That summarizes the very basics of using Visual Studio to run NGen.exe for your application. T article hopefully articulates all the various steps needed to get the environment right, and poin potential pitfalls along the way. We'd love to hear what you think! Have you used Visual Studio past to enable NGen for your application? If not, what do you use? How was your experience? use the comments section below for any feedback and questions.

Pracheeti Nagarkar

CLR Codegen team

NGen: Measuring Warm Startup Performance with Xperf

April 27, 2010, 10:52 am

» Next: NGen: Measuring Working Set with VMMap
« Previous: NGen: Getting Started with NGen in Visual Studio

This is article 2 of 4 in the **NGen: Walkthrough Series**.

This article is part of a series of blog posts intended to help managed code developers analyze if Native Image Generation (NGen) technology provides benefit to their application/library. NGen refers to the process of pre-compiling Microsoft® Intermediate Language (MSIL) executables into machine code prior to execution time.

Startup time is defined as the time it takes for an application from launch to startup such that it is now responsive to user input. It is typically thought of as having two variants, cold startup and warm startup. The time it takes for an application to start up on a machine that has just been booted is typically referred to as cold startup time. The time it takes for the application to start up on its second launch is referred to as warm startup time. The difference between the two is that cold startup time is bound by the need to fetch pages used by the application from disk. In contrast, warm startup is typically only bound by the work the application (and underlying runtime layer) needs to do to start up, since the pages needed by the application under normal circumstances don't need to be fetched from disk.

Using native images does not necessarily shorten cold startup time since the native image files are significantly larger than their corresponding IL files and may take longer to pull from the disk. We will not talk about cold startup time in this article although that may be a topic we address in a future post. Loading native images however, can help shorten application warm startup time since the CLR does not need to run the JIT compiler on the managed assemblies at application launch time. This article is a walk through to help developers use publicly available tools to evaluate how much warm startup benefit the application will see if NGen were to be used.

We will look at two contrasting scenarios to measure warm startup time, the first will involve an application where most of the managed assemblies being loaded have native images and the one will involve the same application where most of the managed assemblies will NOT have na images.

Further Reading

1> Performanc

2> A model for startup-time-of-a

Getting Start

Download Windo

Xperf is an ETW enables drill do application.

After downloading Xperf, you can get access to the tool by launching All Programs -> Windows Command Prompt. This will place Xperf on the path in the Command Prompt window.

Further Reading

1> Event Tracing for Windows (ETW): http://msdn.microsoft.com/en-us/magazine/cc163437.a

2> Additional Information on Xperf: http://blogs.msdn.com/pigscanfly/archive/2008/02/09/xpe tool-in-the-windows-sdk.aspx

Warm Startup Time with Native Images

In order to illustrate the analysis of startup performance, we will use a well known large applica has components written in managed code that have native images; Visual Studio 2010. The sc will launch a basic HelloWorld WPF application in Visual Studio 2010, wait till the UI is responsi shutdown. In order to collect CPU time traces for later analysis, Xperf will be used to enable tra before application launch, and tracing will be stopped after the scenario is run.

The steps below outline the sequence of actions needed.

1> Enable ETW tracing using Xperf

```
xperf -on base+cswitch
```

(Starts a trace with BASE Kernel Group and Context Switch Kernel Flag)

2> Run scenario of interest

```
"C:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE\devenv.exe"
"c:\users\xyz\documents\visual studio 2010\Projects\WpfApplication1\WpfApp1.sln"
```

3> Disable ETW tracing (data is saved to an ETL trace file) using Xperf

```
xperf -d trace4.etl
```

4> View CPU time results using Xperfview

```
xperf trace4.etl
```

In the last step Xperf will forward the request to Performance Analyzer which will open and disp graphical view of the data in the file, similar to what is shown below.

clip_image002

Screenshot 1: Windows Performance Analyzer

Performance Analyzer has several rows of graphical data, the one of interest to us is the row t **Sampling by CPU**. Select the high activity region on the graph (assuming that devenv.exe wa large application launched during this time-frame, the spikes seen in the CPU utilization should to spot), right click in the region and select the **Summary Table** item. (Alternatively, you can al at the **CPU Sampling by Process** row, and in the **Processes** drop down menu, you can sele **devenv.exe** to see the activity for that process.)

The **Summary Table** has tabular CPU utilization data for all the processes running on the mac that time window; **devenv.exe** should be part of this list. Under the **%Weight** column you will time taken by devenv.exe on the machine as a **percentage of the total CPU utilization on machine**.

Expanding the devenv.exe process displays a view of the percentage of CPU time spent in eac module loaded within this application. Notice in Screenshot 2 below that the bulk of the time wa in clr.dll (20.96% total across the machine, which means that it was ~30% of the time spent witl process devenv.exe) and clrjit.dll (8.68% total across the machine, which means that it was ~1 within the process). Note that some time was spent in the JIT even though most of this scenari running using native images. The use of native images can be seen in the fact that *.ni.dll files listed in the module list (mscorlib.ni.dll, system.xaml.ni.dll etc).

clip_image004

Screenshot 2: Windows Performance Analyzer, CPU Utilization Summary Table drilldown with images present

Further Reading

1> Detailed guidance on using Performance Analyzer: All Programs -> Windows Perfo Toolkit -> Windows Performance Toolkit Help. Select Quick Start Guide: WPF Basic Detailed Walkthrough.

Warm Startup Time without Native Images

Now let's turn our attention to what the startup time characteristics look like when native image used in the Visu generated by ur recommended t previously, to ge

The increase in we can see that in devenv.exe w of the time spen total, which mea increase from th increase! Also n extension, indic

This large increase in time spent in clr.dll and clrjit.dll is to be expected when native images ar present. Since native code does not exist for many assemblies, the CLR needs to invoke the J compile methods to native code at runtime, explaining the increase in time spent in clrjit.dll. Tir in clr.dll also increases because the CLR needs to execute code to invoke the JIT, and it needs create internal data structures that the native code will need.

clip_image006

Screenshot 3: Windows Performance Analyzer, CPU Utilization Summary Table drilldown witho images present

Wrapping it up!

This article demonstrated how to measure an application's warm startup time using Xperf, and the benefit of using native images for large applications. Hopefully, the information above will h do a first round of evaluation of whether your managed application/library will benefit from the NGen. We'd love to hear what you think! Have you used Xperf before to do similar analysis? W the pitfalls you ran into? Please use the comments section below for any feedback, questions you'd like to share.

Lakshan Fernando, Pracheeti Nagarkar

CLR CodeGen team

NGen: Measuring Working Set with VMMap

April 27, 2010, 10:59 am

≫ Next: NGen: Creating Setup Projects
≪ Previous: NGen: Measuring Warm Startup Performance with Xperf

This is article 3 of 4 in the **NGen: Walk-through Series**.

This article is part of a series of blog posts intended to help managed code developers analyze if Native Image Generation (NGen) technology provides benefit to their application/library. NGen refers to the process of pre-compiling Microsoft® Intermediate Language (MSIL) executables into machine code prior to execution time.

Working set is the amount of physical memory that has been assigned by the operating system to a given process. For managed applications, NGen helps to reduce the working set in 2 ways: the application will not need to load the JIT into the process (process specific benefit), and the native image for a library will be shared across multiple managed applications running at the same time (machine wide benefit). As with everything performance related, you can only decide whether using NGen benefits working set for your application by measuring it. This article will walk through how to perform such measurements and what to watch for.

This article contains the following sections: *Getting Started with VMMap, The Basics: Is the JIT getting loaded?, The Basics: Using the GAC, Impact of Base Address Collisions (Rebasing): Pre-Vista, Impact of Base Address Collisions (Rebasing): What about Vista?, Cross-Process Sharing of Native Images, Wrapping it up!*

Getting Started with VMMap

Download: http://technet.microsoft.com/en-us/sysinternals/dd535533.aspx

VMMap is a handy tool that provides visualization for process memory usage. In order to easily launch it on a currently running process, use the command "vmmap.exe –p <name_of_process_exe>".

VMMap_Table1

Screenshot 1: Overall View

In Screenshot 1 above, observe the various memory types listed in the rows in the upper pane
Private, Shareable etc.). For each memory type (for example, Private), there are several colur
detail how that memory type breaks down; how much is the Committed memory, Total Working
Private Working Set, Shareable Working Set etc. Note that the precise definition of each of the
available from the Help->Quick Help menu in the UI. The Image memory type is easier to use f
tracking purposes because it maps to the executable file that launched the current process be
analyzed. It also
of memory they
reduce the Priva

In order to see t
named Image. T
resource usage

Vmmap_Sc

Screenshot 2: I

For each file (fil
resource usage
Shareable WS a
Shareable and S

Let's understand one of the rows shown above, for the file mscorlib.ni.dll.

Size = 13,936K. This refers to the size of the image which comes from the PE header of the file
curious, to see where this number comes from, do the following: From a Visual Studio SDK cor
prompt use "link.exe /dump /headers <PathToFile>", and look under the Optional Header Value
"size of image" in hex.

Committed = 13,936K. This is the amount of allocation backed by virtual memory.

Total WS = 788K. This is the amount of physical memory that is assigned to this file.

Private WS = 64 K. Of the total physical memory assigned to this file, this amount cannot be sh
with other simultaneously running processes that also need this file.

Shareable WS=724K. Of the total physical memory assigned to this file, this amount could be s
with other simultaneously running processes that also need this file.

Shared WS = 684K. Of the total Shareable working set, this is the amount that is currently beir
shared with another process that also needs this file. If there were no other process that need
this number could be 0.

Supported Environments: Windows XP and newer Operating Systems. 32 bit and 64 bit.

Further Reading

1> VMMap: http://technet.microsoft.com/en-us/sysinternals/dd535533.aspx

The Basics: Is the JIT getting loaded?

One of the quickest ways to reduce the total working set of a managed application is to ensure
only essential modules get loaded into the process.

If the application and its entire closure of dependencies has been NGen'd, in the general case
expected that the JIT will not get loaded. In the Details column in the lower pane of VMMap, ch
clrjit.dll (.Net Framework v4.0) or mscorjit.dll (Pre-.Net Framework v4.0) has not been loaded.
been loaded, it indicates that something within your application is getting JIT-ed. This may be a
dependency that was not NGen'd or it may be a few methods within an assembly for which nati
could not get generated. For the former case, Fusion Log Viewer can be used to determine th
assembly for which a native image could not be found. For the latter case, the JIT Managed D
Assistant helps answer the question around which method the JIT is getting invoked for. See th
below in Further Reading for details on how these tools can be used.

Working Set Impact: When the JIT is loaded into the process, it contributes ~200K to the Total
does not seem like a whole lot when looked at independently. However, in order to invoke the J
CLR's engine itself executes more code and as a result more pages from the engine's DLL (ca
clr.dll in .Net Framework 4.0 or called mscorwks.dll in prior releases) are pulled into the Total W
example I used, the Total WS of clr.dll went up by ~100K. That's not all. Since a particular asse
A.dll (or a part of it) needed to get JIT compiled, the IL for that assembly also needed to be loa
Depending on the size of this assembly (or the size of the method that needed to be JIT compi
IL also contributes to the Total WS. Moreover, since the JIT compiled code cannot be shared a
process boundaries, if A.dll was also needed subsequently by another process on the machine
new process would need to incur its own Total WS cost to JIT compile the code.

Of course, if the size of the assembly (or method within the assembly) being JIT compiled is sm
may not care about this working set hit. Doing your own measurements will help you reach a d

Further Reading

1> Fusion Log Viewer: http://msdn.microsoft.com/en-us/library/e74a18c4.aspx

2> JitCompilationStart Managed Debugging Assistant (MDA): http://msdn.microsoft.com/en-us/library/fw872k46(VS.80).aspx

The Basics: U

In some cases, possible that bo the application's the IL and the se below. Note that the type C:\Win

Vmmap_Sc

Screenshot 3: F

However, for any the native image and native imag being loaded liv mixed mode ass stop the additional load of the IL. Again, it is important to keep in mind that the additional load assembly depends on the size of the assembly itself – it may not be large enough to make you install the assembly into the GAC.

Aside: Speaking of the GAC, prior to .Net Framework 3.5 SP1, it was strongly recommended th strong name signed libraries used by an application be installed into the GAC. The reason was for strong name signed assemblies outside the GAC, the CLR would need to load the entire IL assembly and compute a hash over its contents and compare it with the assembly's signature permitting the load of the native image for security reasons. Starting with .Net Framework 3.5 S feature called strong name bypass eliminates the need to compute this hash for full trust case Screenshot 3, the assembly foo.dll is strong name signed and outside the GAC. The IL still get mapped and contributes to Total WS, but we do not incur an additional working set hit from clr. computing the strong name hash. As an experiment, if the strong name bypass feature is disal working set for clr.dll will be higher.

Further Reading

1> Strong Name Bypass: http://blogs.msdn.com/shawnfa/archive/2008/05/14/strong-name-bypass.aspx

Impact of Base Address Collisions (Rebasing): Pre-Vista

When rebasing occurs, the Private WS increases and the Shareable WS decreases; this is no for machine-wide memory usage. Let's look at why rebasing affects working set.

Native Images typically get loaded at an address that is specified in its PE Header. From a Visu Command Prompt, do a "link.exe /dump /headers <PathToNativeImageFile>". Screenshot 4 be shows this.

Link_ScreenShot4

Screenshot 4: Preferred Base Address

So who chooses the base address? Unless you specify a base address during compile time (fo instance, via the /baseaddress option to csc.exe), the CLR will pick one! If a base address was specified during compile time, the compiler assigns a default base address of 0x4000000. Whe creating a native image, the CLR translates this default address for an executable to 0x30000 for a DLL to 0x31000000. Screenshot 5 shows the DLL default above, as picked by the CLR. T address is the one used by the operating system to load the PE file at. VMMap shows which ad was used in the very first column "Address" in the lower pane.

Naturally, if there are multiple managed DLLs loaded into the process with the same preferred address assigned, not all can be loaded at the same address. The first DLL native image to be will occupy 0x31000000 and all subsequent DLL native images will get loaded at other address is called rebasing. The SDK tool Fusion Log Viewer will log when a native image gets rebased should expect to see a message similar to the one shown in Screenshot 5 below, if rebasing oc

Screenshot6_Fuslogvw

Screenshot 5: Fusion Log Viewer indicating rebasing of a native image

Let's take a quick look at why rebasing is bad. Each native image contains absolute addresses references to items (like strings) within the native image itself. If the native image gets loaded a address other than the preferred base address, the CLR needs to adjust those references (al as performing fixups) – this is done by writing to the page that contains the reference, thereby private pages that cannot be shared with other processes that might also want to load the sam image. Using VMMap, when rebasing occurs, the Private WS number increases and the Share decreases. This is not optimal for total machine wide memory usage.

The use of hard binding further complicates this analysis. Note that if the managed application hard binding for native images, when the application launches, the hard bound dependencies loaded first before the application. In case any of the hard bound dependencies gets rebased dependency itself will need to have fix ups, creating private pages. However, the application's image that stores references to this dependency native image will need fixups as well. The app Private WS will increase for two reasons – once for the rebased dependency and once for the assembly that depends on this rebased dependency.

Remember when using VMMap: For Total WS and Private WS – lower is better. For Shareable higher is better.

Further Reading

1> NGen and Rebasing: http://msdn.microsoft.com/en-us/magazine/cc163610.aspx#S5

2> Hard Binding of Native Images: http://msdn.microsoft.com/en-us/magazine/cc163610.asp›

Impact of Ba

Starting with .Ne
native images a
Randomization).
header of a nati
preferred base

As mentioned a
base address c

Further Reading

1> Address Sp
us/magazine/2007.04.vistakernel.aspx

Cross-Process Sharing of Native Images

One of the benefits of having native images is that they can be shared across multiple manag
processes on a machine. This is particularly helpful for server scenarios where there may be r
managed applications running on the machine, that depend on a common set of managed libr
an application has already loaded a native image, and a second application is started that use
same native images, the pages from the native image that are still marked shareable, will be s
The second application will not incur a working set hit for those pages.

Let's look at this using VMMap.

Screenshot 6: Shareable WS becomes Shared WS with multiple managed applications

Assume that Dep1.dll is our library that could potentially be shared. In VMMap, for the native i
foo.dll, the "Shareable WS" is listed as 8K, and the "Shared WS" is nothing. This indicates tha
chunk of memory in the native image for Dep1.dll is available for sharing, in case another proc
wanted to share it. When we launch a second application that also uses Dep1.dll, hit the Refre
option in VMMap to show the new working-set numbers. We'll now see that the "Shared WS" n
8K. This indicates that of the memory from Dep1.ni.dll that was available for sharing, we share

Wrapping it up!

That summarizes the very basics of using VMMap to analyze the impact of NGen on the workir
a managed application. This article hopefully articulates how VMMap can be used and points
to watch for during the measurements. We'd love to hear what you think! Have you used VMM
past to analyze the impact of NGen? If not, what do you use? How was your experience? Pleas
the comments section below for any feedback, questions and tips you'd like to share.

Pracheeti Nagarkar

CLR Codegen team

NGen: Creating Setup Projects

April 27, 2010, 11:05 am

≫ Next: JIT ETW Tail Call Event Fail Reasons
≪ Previous: NGen: Measuring Working Set with VMMap

This is article 4 of 4 in the **NGen: Walk-through Series**.

The NGen technology is designed to be used during the installation phase of a managed application or library. This article will talk about the various installer technologies available, which one to choose, and how to invoke NGen given that installer technology.

Installer Toolsets

The fundamental thing to know before we take a look at installers is that NGen is a tool that can be run only with administrator privileges. A non-admin user cannot invoke NGen.exe. This means that any installer technology that cannot run with administrator privileges cannot be used to invoke NGen easily. Non-admin installer technologies like ClickOnce sometimes use an MSI wrapper to invoke administrator-only actions like NGen.

There are several tools available to create a Windows Installer file (MSI file) – Visual Studio Setup and Deployment Projects, Install Shield 2010 Limited Edition, Windows Installer XML Toolset (WiX), etc. This MSDN article gives a breakdown of how these tools compare: http://msdn.microsoft.com/en-us/library/ee721500(v=VS.100).aspx. For a simple project which just installs a few binaries, the Visual Studio Deployment Project is easy to ramp up on and can quickly produce a workable package. However, for production quality applications, WiX tends to be the installer toolset of choice. Among other benefits, WiX has MSBuild support (which implies you don't need to install Visual Studio in order to create a setup package) and stores all data in XML files (which makes it easily editable).

The rest of this article will focus on using NGen via the WiX toolset.

Further Reading

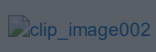
1> Using NGe
us/library/3hwzz

Getting Start

Assuming you h
http://wix.source

This version is c
2010 RTM as w
release soon.

The walk throug
called *HelloWinForm* exists already. To create the WiX project that will install this application, ir
Visual Studio solution, in **Solution Explorer** right click on the solution, **Add** a **New Project** a
the **Windows Installer XML** template select the **Setup Project**.



Screenshot 1: Selecting a Setup Project

Next, in Solution Explorer, under *WixProject1* right click on the **References** node and choose **Reference**. In the **Projects** tab, select the project named *HelloWinForm*. This will ensure tha *HelloWinForm* project will build prior to the *WixProject1*, and the input to *WixProject1* is the ou the *HelloWinForm* project.

WixProject1 will now contain a default .WXS file which will need some modifications before you successfully. This XML file will have several TODOs that will need to be adjusted. Under the Co tag, add a File tag which looks like this –

```
<FileId="WindowsFormsApplication1File"Name="$(var.HelloWinform.TargetFileName)"Sourc
```

Further Reading

1> Getting the latest updates for WiX: http://wix.sourceforge.net/

2> Using WiX in Visual Studio to create a basic setup package: All Programs -> Windows Inst Toolset 3.5 -> WiX Documentation. Go to Using WiX in Visual Studio -> Creating a Simple Setu

Adding NGen steps to the WiX Project

Now that we have a basic WiX project working without NGen, the next phase is to incorporate N this project so we can NGen the binaries produced in the *HelloWinForm* project.

In Solution Explorer, right click on the **References** node and choose **Add Reference.** In the tab, find **WixNetFxExtension.dll** under the bin directory (the full path will likely be something C:\Program Files\Windows Installer XML v3.5\bin\WixNetFxExtension.dll), and click **Add.** Add a namespace called netfx by changing the Wix tag in the WXS file as shown below.

Before:

```
<Wixxmlns="http://schemas.microsoft.com/wix/2006/wi">
```

After:

```
<Wixxmlns="http://schemas.microsoft.com/wix/2006/wi"xmlns:netf
="http://schemas.microsoft.com/wix/NetFxExtension">
```

This will enable intellisense for the netfx namespace.

The next step is to enable NGen installation for the binary that is built in the *HelloWinForm* prc do that, add the following line under the File element.

```
<netfx:NativeImageId="WindowsFormsApplication1File.exe"Platform="32bit"Priority="1"D
/>
```

Detailed documentation for the WiX schemas can be found via the Windows start menu, All Pr > Windows Installer XML Toolset 3.5 -> WiX Documentation. Information about the NetFx name lives under Windows Installer XML (WiX) Help -> WiX Schema References -> Netfx Schema -> NativeImage Element (Netfx Extension).

The resulting WXS file will look as follows –

```
<?xmlversion="1.0"encoding="UTF-8"?>

<Wixxmlns="http://schemas.microsoft.com/wix/2006/wi"xmlns:netfx
="http://schemas.microsoft.com/wix/NetFxExtension">

  <ProductId="3645a635-5cb6-48da-8a2a-
6d9bd9778aee"Name="Wix35Project1"Language="1033"Version="1.0.0.0"Manufacturer="Wix35
b331-4ca3-94b1-2bfd652a280a">

    <PackageInstallerVersion="200"Compressed="yes" />
```

```
    <MediaId="1"Cabinet="media1.cab"EmbedCab="yes" />


    <DirectoryId="TARGETDIR"Name="SourceDir">
        <DirectoryId="ProgramFilesFolder">
            <DirectoryId="INSTALLLOCATION"Name="Wix35Project1">
                <ComponentId="ProductComponent"Guid="13b3ea27-e675-4e39-b225-62b67d2
<FileId="Windo
<netfx:NativeI
/>
                </
        </Dire
    </Directory
    <FeatureId
        <Compo
    </Feature>
  </Product>
</Wix>
```


Specifying a priority of 0 ensures that the native image compilation will be done synchronously the installer is running. Specifying a priority value of 1, 2 or 3 will make the compilation occur i background. Priority 1 has the effect that assemblies are immediately compiled in the backgro priority 2 means the assemblies will be compiled after the priority 1 assemblies are done, and assemblies are compiled at machine idle time. If no priority value is specified, the priority will d be 3. The WiX toolset will issue an "ngen install" or "ngen uninstall" action depending on whet resulting MSI is being used to install the application or remove it from the machine. The toolse always issue an "ngen update /queue" as the last command, in order to trigger the NGen serv recompile any native images that may be out of date.

Further Reading

1> How to NGen files in an MSI-based setup package using WiX: https://blogs.msdn.com/astebner/archive/2007/03/03/how-to-ngen-files-in-an-msi-based-setup package-using-wix.aspx

2> Introducing NGen Support in WiX: http://installing.blogspot.com/2006/06/ngen-support-in-

3> Synchronous versus Asynchronous Native Image compilation: http://msdn.microsoft.com/e us/library/ms165074(v=VS.80).aspx

4> Installing file into the GAC using WiX: http://blogs.msdn.com/astebner/archive/2007/06/21/3450539.aspx

Wrapping it up!

That summarizes the very basics of using the Windows Installer XML toolset to invoke NGen.e installer for your application. This article hopefully articulates the various steps needed to get environment setup, and points out nuances along the way. We'd love to hear what you think! H used WiX in the past to enable NGen in your installer? If not, what technology do you use? Ho your experience? Please use the comments section below for any feedback and questions.

Janine Zhang, Pracheeti Nagarkar

CLR Team

JIT ETW Tail Call Event Fail Reasons

May 7, 2010, 2:48 pm

» Next: Testing, Testing: Hey, is this thing on?
« Previous: NGen: Creating Setup Projects

This is a follow-up post for JIT ETW tracing in .NET Framework 4. These are some of the possible strings that might show up in the FailReason field of the MethodJitTailCallFailed event. These are reasons that come from or are checked for by the VM (as compared to the JIT) and are listed in no particular order:

- "Caller is ComImport .cctor" - This means the caller is a static class constructor for a type which has a base type somewhere in the class hierarchy marked with the ComImportAttribute. This is caused by an implementation choice within the runtime for managed objects that effectively derive from native COM objects. You must remove the attribute if you want to perform a tail call.
- "Caller has declarative security" - This means the caller has a declarative security attribute applied to it (usually an Assert or a Demand, but Deny and PermitOnly also prevent tail calls). The current implementation relies on the caller remaining on the stack to enforce the security attribute. You must remove the attribute from the caller, if you want to perform a tail call.
- "Different security" - The caller and callee have different permissions, and the one with the 'lower' permissions must remain on the stack. Since comparing permissions is expensive, we simplify it to Full Trust and non-Full Trust. Full Trust code can do anything, including tail calls. One other special case is homogenous appdomains, where everything in the

appdomain has the same permissions, so even if the callee is unknown (due to virtual or in
calls), the callee must have the same permissions as the caller. If you want to do a tail call,
homogenous appdomain, grant the caller Full Trust, or put the caller and the callee in the s
assembly and make sure it is a direct call.

- "Caller is the entry point" - If there is no "tail." instruction prefix, the JIT is not allowed to ger
tail call from a method marked as the entrypoint for a module. The idea is that programers
see their Ma
- "Caller is ma
marked with
method fram
tail calls from
or remove th
- "Callee migh
stack walk to
tail calls. This
generate tail
- "Caller is a C

From x86 JIT, we

- "Caller is syn
needs to lea
release the r
- "Caller is var
about varargs in C# (**not** the params keyword), search for __arglist.
- "Caller requires a security check." - The caller is marked with mdRequireSecObject for impe
security. With our current implementation, such methods need their own call frame so the
corresponding Assert or Deny will end at the return of the method. If you want to do a tail c
remove the imperative security calls.
- "Needs security check" - Same as above.
- "Callee is native" - We currently cannot tail call from managed code to native code.
- "PInvoke calli" - Same as above.
- "Return types don't match" - The caller and callee must have the exact same return type. I
want to do a tail call, change the return types to match.
- "Localloc used" - This is just an implementation limitation of the x86 JIT. In C# if you use sta
then the JIT cannot be sure of the intended lifetime, and so it goes safe and prevents the ta
you want to do a tail call, remove the stackalloc.
- "Need to copy return buffer" - If the return value doesn't fit in a register, the caller needs to
a buffer. Normally the JIT reuses the caller's return buffer for the caller to avoid a copy, but
sometimes it can't, and because it now has to do a copy after the callee returns, it can't do
call. If you want to do a tail call, use an out parameter rather than a return value.
- "Changed into handle" - The C# expression 'typeof(XXX).TypeHandle' involves a call to the
method get_TypeHandle. The JIT can turn that whole expression (including the call) into a
embedded constant (the TypeHandle as provided by the VM). We think that is faster and b
than any tail call.

From the 64-bit JIT, we get this list of failure reasons:

- "function has EH" - The IA64 JIT doesn't support tail calls from methods with try/catch/finally
unless the call uses the "tail." instruction prefix. If you want to do a tail call remove the exce
handling clauses or add a "tail." prefix.
- "found symbol with address taken" - if the call doesn't use the "tail." instruction prefix and th
method takes the address of a local, the JIT doesn't do enough analysis to see if it is addre
escaped (meaning the callee uses the address to access the caller's local) and so it just do
to optimize a normal call into a tail call.
- "local address taken" - Same as above.
- "synchronized" - This is the same as the x86 JIT's "Caller is synchronized".
- "caller's imperative security" - This is the same as the x86 JIT's "Caller requires a security c
- "caller's declarative security" - This is the same as the VM's "Caller has declarative security
- "not optimizing" - The JIT disabled all optimizations, and so it only performs a tail call if the "
prefix is present. If you want to do a tail call, either add the "tail." prefix or re-enable optimiz
Some of the reasons why JIT optimizations might be disabled include: using
MethodImplOptions.NoOptimization, a method that is too big or too complex to optimize, run
under a debugger, and certain compiler switches.
- "localloc" - This is the same as the x86 JIT's "Localloc used", except the 64-bit JIT will do a
the call explicitly uses the "tail." instruction prefix.
- "GS" - The method uses local buffers (unmanaged arrays) and the JIT adds extra code to d
buffer overruns before they can be exploited. These extra checks are incompatible with tail
our current implementation. The name comes from the C++ compiler's /GS command-line s
and attempts to prevent many similar issues as they appear in unsafe managed code.
- "turned into intrinsic" - The 64-bit JIT cannot tail call certain methods that effectively turn int
code. This is similar to the x86 JIT's "Changed into handle".
- "P/Invoke" - This is the same as the x86 JIT's "Callee is native".
- "return type mismatch" - The caller and callee must have compatible return types (types tha
require any conversion at the hardware level). This is similar to the x86 JIT's "Return types
match", but the 64-bit JIT is slightly more permissive.
- "processor specific reasons" - The caller and callee's signature are different enough that th
convention makes it hard (or impossible) to do a traditional optimized tail call. This is usuall
by the callee having more (or bigger) arguments than the caller. On x64 if the "tail." instruct
is used, the JIT will generate a HelperAssistedTailCall.

It is worth noting that the 64-bit JIT tries to optimize almost all calls into tail calls. The JIT also i
certain amount of knowledge, intent and analysis when the "tail." IL prefix is used on a call. A r
call (no prefix) is sort of like telling the JIT to make a call however it deems best. The JIT then
some quick conservative checks to see if a tail call is possible and would be as good as or bet
a normal call. On the other hand a call with the "tail." prefix is sort of like telling the JIT to try a
possible to make a tail call, because the programmer or the compiler did some big analysis an
that, despite what the JIT might think, the tail call is safe and will be better than a regular call.
only things the JIT has to check for are known problems (verification, security, and implementa
limitations).

The x86 JIT, on the other hand, currently only attempts to do a tail call when the IL explicitly us
"tail." prefix. Thus the x86 JIT only checks for correctness.

It is my understanding that the C# and VB.NET compilers never emit the "tail." instruction prefi
C++ and F# compilers generate it automatically, so the programmer has very little control over
condition. So unless you write in IL, or use some form of IL rewriter, your ability to add or remo
".tail" prefix is limited at best.

Lastly if you're still reading you have probably noticed that there is a lot of redundancy. This i
because the messages are generated by different components in the runtime - the VM, the x8

the x64 JIT - which were developed, and have evolved, fairly independently. There is also son amount of redundancy as a safety precaution.

Grant RIchins
CLR Codegen Team

Tes
on
Septemt
≫ Nex
≪ Pre

Hey fo
aficion
hasn't
workin
tanger
"by de
details

Kevin Frei

Dev Lead for the .NET Codegen team

RyuJIT CTP1 is available for public consumption!

September 30, 2013, 2:06 pm

≫ Next: Floating Point: A Dark & Scary Corner
≪ Previous: Testing, Testing: Hey, is this thing on?

http://blogs.msdn.com/b/dotnet/archive/2013/09/30/ryujit-the-next-generation-jit-compiler.aspx

'nuff said (for now).

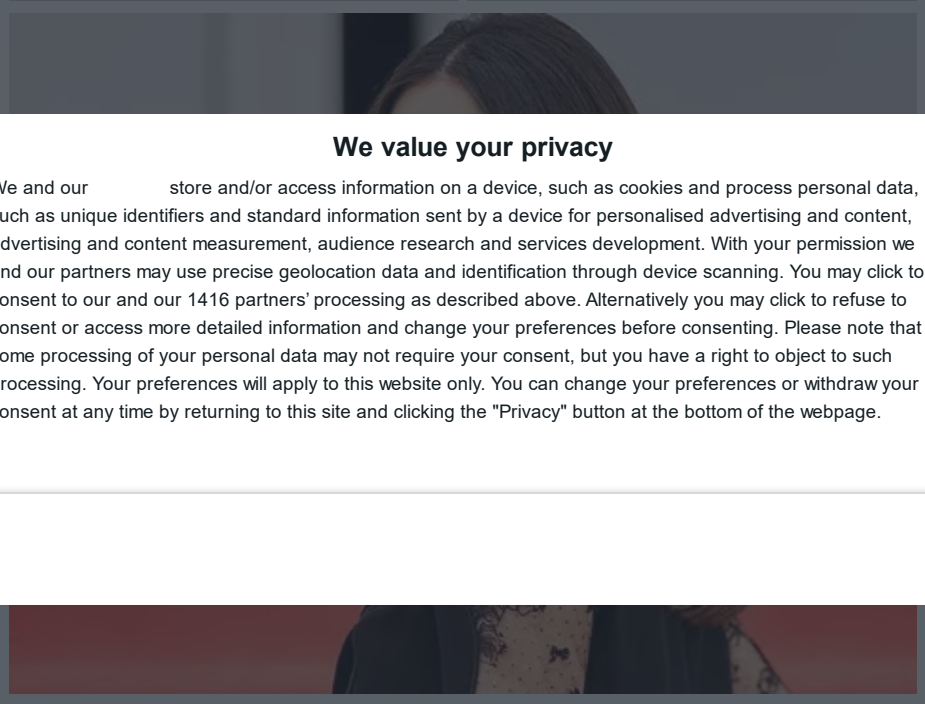
-Kev

If Looks Could Kill, These Women Would Be On Top

Brainberries

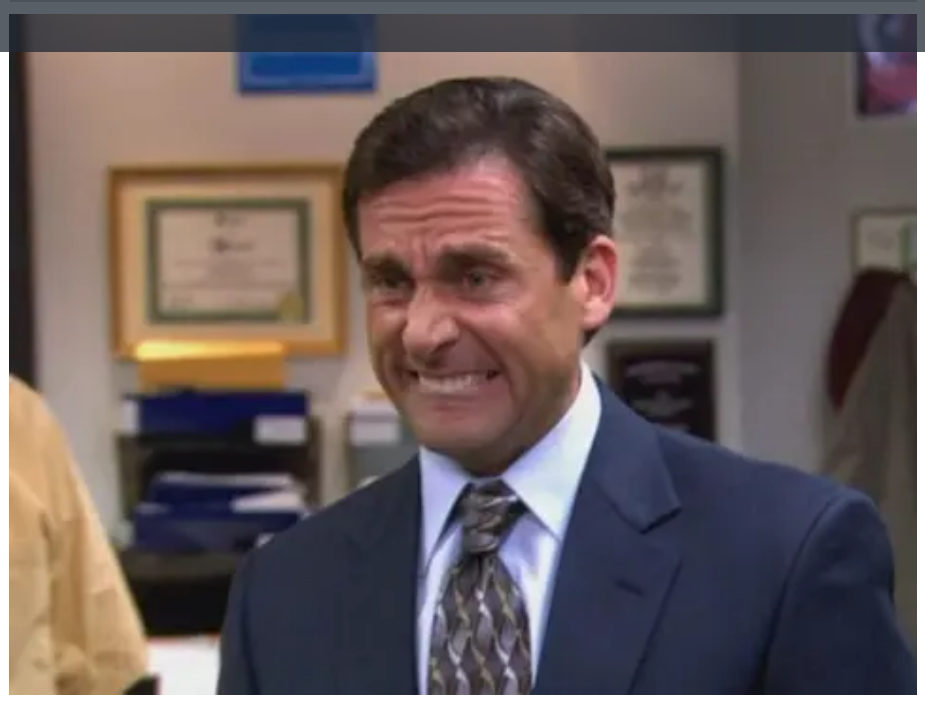
A Rihanna Museum Is Probably Opening Soon

Brainberries



The 10 Most Stunning Women From Lebanon - Who Is Your Favorite?

Brainberries

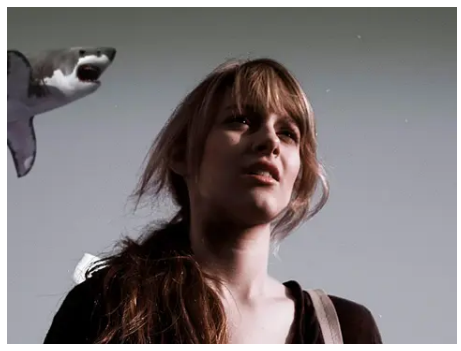


Mystery Solved: Here's Why These 9 Actors Left Their TV Shows

Brainberries

Will You Survive? 10 Things To Keep In Your Emergency Kit

Brainberries

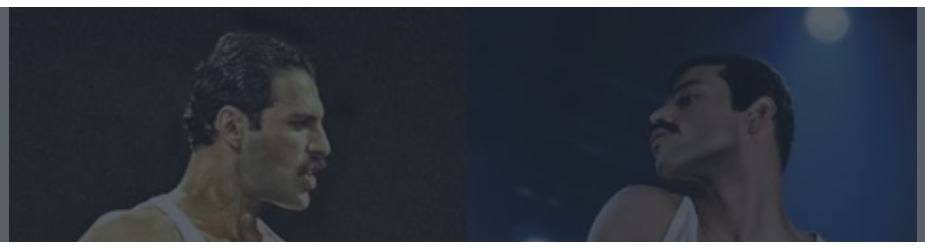


Films To Make You Question Everything You Know About Cinema

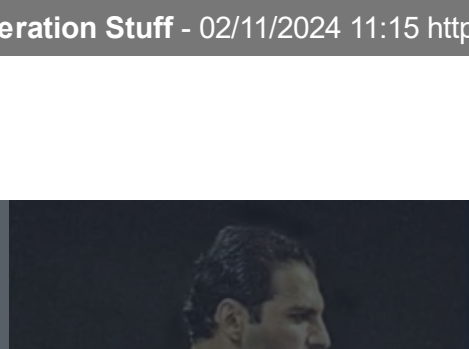
Brainberries

Critics Were Impressed By The Way She Portrayed Grace Kelly

Brainberries

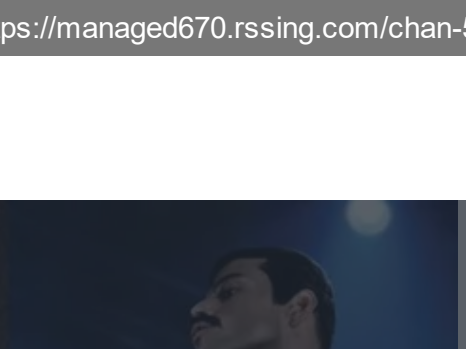


Brainberries



Where Are They Now? 9 Ex-Actors Found Unexpected Career Paths

Brainberries



And They Did Show This In Bohemian Rapsody!

Brainberries

Tarantino's Latest Effort Will Probably Be His Best To Date

Brainberries

Top 10 Pop Divas (She's Not Number 1)

Brainberries