

Home ▾ Search ▾ Register RSS ▾ Embed RSS ▾ Super RSS ▾ Contact Us ▾ Login ▾

RSSING>> ▾ LATEST ▾ POPULAR ▾ TOP RATED ▾ TRENDING ▾ english 🔍

+ ★ Channel: JIT, NGen, and other Managed Code Generation Stuff

Viewing all 40 articles Page 1 ▶ Remove ADS

Browse latest View live

## 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.

search RSSing.com... Search

## To NGen or Not to NGen?

September 15, 2007, 3:19 am

» Next: Running NGen as part of installing a Microsoft Exchange...  
« Previous: Welcome to the CLR Code Generation Team's blog

search RSSir Search

### TOP-RATED IMAGE < >

Hamburg, which was famous for its grand sauerkraut. German immigrant introduced the hamburger sandwich. Then, the introduction of the bun was owner. American entrepreneur of McDonald's one of world's largest fast food chains can find them in Japan, Germany, France, Great Britain, Australia, even Moscow. In fact, there are 13,000 in 120 countries. There's even at Louis, Mississippi, and another in Alaska. The first McDonald's was opened is now a museum.

What is most American fast food is popular?

Lee Lee, from South Korea, "I often go to McDonald's with my mother and father," says. "I like the hamburgers with cheese and onions." **Fast Food Reading Worksheet**

Natalia Petrenko, a girl from Moscow, says "I find hamburgers with cheese and onions are very nice."

Karen Smith, from Frankfurt: "McDonald's is our favorite meal and talk for hours in a friendly atmosphere. There are lots of young people and their milkshakes, yum..."

Oliver is very critical of this sort of food. Lots of your overweight and parents blame these high-calorie foods that their children eat.

A. Say if the following sentences are true (T) or false (F). Guess  
1. The hamburger is originally American.

### LATEST IMAGES

Democrats linked to 'pro-choice' GOP candidates

October 4, 2024, 3:19 pm



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 NGen-ed because the time spent JIT-compiling the code is typically a small portion of the overall start up time. As a matter of fact, since NGen images are substantially bigger in size than the corresponding IL assemblies, using NGen might actually result in increased disk I/O and hurt cold start up time.
- Server applications: Server applications that aren't sensitive to long start up times, and don't have shared components are unlikely to benefit significantly from NGen. In such cases, the cost of using NGen (more on this below) may not be worth the benefits. SQL-CLR for example, isn't NGen-ed.

If it seems that your application is likely to benefit from NGen, here are a few things you might want to keep in mind:

1. NGen is triggered by issuing commands at install time to the ngen.exe tool in the .NET Framework

redistributable. To find out more about hooking NGen up, see here:

<http://blogs.msdn.com/astebner/archive/2007/03/03/ngen-files-in-an-msi-based-setup-package-using-wix>

*Note: Currently you need admin privileges to issue commands to ngen.exe.*

2. Just running ngen.exe on your assemblies and then re-running your performance scenarios to measure the wins is very not going to give you a true indication of the wins. Basically takes some effort to get the best performance out of it. Detailed guidelines on how to get the best performance out of NGen, please take a look at this article in the MSDN magazine:  
<http://msdn.microsoft.com/msdnmag/issues/06/05/csharp>
3. Using NGen has implications for servicing your application. Patch installer will need to issue NGen commands to ensure that the images invalidated by the patch get regenerated. .NET Framework 2.0 supports an "ngen.exe update /queue" command that should work well in most cases (this will regenerate all invalid NGen images at machine idle time). Specific assemblies are perf-critical and need to have their NGen images immediately after the patch install. Issue separate NGen commands for those ("ngen.exe update /queue <CriticalDLL>"), followed by the "ngen.exe update /queue" command.
4. Using NGen implies your application's disk footprint will increase (NGen images are fairly big when compared to regular assemblies).
5. In a handful of scenarios, cold start up and/or application execution can get slower when you use NGen, so you definitely want to measure the performance with vs. without NGen for all scenarios, and validate that it's a win overall for your application to be using NGen.

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

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

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/WindowsForms>

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 usi...](#)

[« 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=8.0.681.0,  
Culture=neutral, PublicKeyToken=31bf3856ad364e35 is up to date.

Around the same time, one of our developers was helping out on Certificate Revocation List (CRL) issue report by [another customer](#) asked the CSS engineer to check the "Software Restriction Policies" (<http://technet.microsoft.com/en-us/library/bb457006.aspx#EAA>) enabled on the customer's machine. Sure enough, the delay was an Internet Explorer setting called "Check for publisher's certificate revocation" (Options, Advanced, Security) that was checked on the customer's machine, and was resulting in network requests to [crl.microsoft.com](http://crl.microsoft.com) and look up the certificate revocation list at NG. The customer's machine was behind a locked down firewall and could not have access to the internet and the reason for the 10-14 delay for the assembly. This option was unchecked in the IE setting and the issue was reduced to the expected 10 minutes from the 2-hour time. It was considered safe to uncheck this security option in IE since the machine was in a tightly controlled environment.

Short summary of what we learned from this experience: Issuing a synchronous NGen update command in the patch installer may not be the best option. It might be better to schedule an NGen update for machine time using "ngen.exe update /queue", and to (optionally) issue separate NGen install commands for perf-critical assemblies whose NGen images need to be regenerated right away.

---

## 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? W...](#)

[« Previous: Running NGen as part of installing a Microsoft Exch...](#)

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 generates.

Instead it is basic assembly code that does not have any optimizations in it. All local variables references are references into the local stack frame because in Debug code the JIT does not register any local variables. A property of Debug code is that **nop** instructions are inserted before some 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 methods are well optimized. And if they are not optimized as well as you like 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 get 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** to [pdb-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 create 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 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 Explorer), select the 'Build' tab and click the 'Advanced...' button all the way at the bottom (you may need to scroll to see it). In the dialog box that comes up there will be a line 'Debug Info'. This needs to be set to 'pdb-only'. This tells the compiler to still compile with optimizations, but to also create the pdb file.

3. Configure the Debugging Options in Visual Studio to allow the JIT to generate optimized code and to allow you to debug the optimized code.

Go to **Tools=>Options => Debugging => General**

- Make sure that box labeled '[Suppress JIT optimization module load](#)' is Unchecked.
- Make sure that the box labeled '[Enable Just My Code](#)' is Unchecked.

*More information on why this is necessary:*

The Visual Studio IDE makes debugging optimized code harder than you would like.

Whenever you launch a managed program under Visual Studio (Start-Debugging or F5), it will by default, force the JIT to create Debug code. This is true even when you have selected the 'Release' configuration. The reason for this is to improve the debugging experience, but it also makes it impossible to see the Optimized code that you will get whenever your program is not running under the Visual Studio debugger. Your alternative of launching the program using 'Without Debugging' or Ctrl+F5 does allow the JIT to create Optimized code but Visual Studio won't set any of your break points in your optimized code. Thus you normally won't be able to stop and examine the assembly code for a method. We actually want do some debugging with the Optimized JIT code.

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

Note that these are global settings as they affect all solutions; however I found I don't miss any of these 'features'. That is because any code that is compiled for the 'Debug' configuration is still not optimized by the JIT and will still get good debugging there.[\[\\*\]](#)

In my next blog entry I will demonstrate some of the optimizations that are performed in the JIT compiler.

---

[\[\\*\]](#) Note that much of this article was adapted from an earlier article by Varun Morrison ([What does foreach 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 JI...](#)

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 this field is referenced. (Note that this time the pointer points to a location on the heap, instead of the stack.)

**Common operations on value types**

There are only a few common operations on value types. Here is what they are implemented internally.

- Field access

Given that a value type instance is referenced by the pointer to the beginning of its blob, accessing its fields is nothing more than adjusting this pointer with the corresponding field offset. For example, if a value type local variable *p* of type PointStruct lives at [EBP-8], then a “mov eax, [EBP-8]” instruction reads the field *x*, and a “mov [EBP-4], eax” instruction writes the field *y*.

- Initialization

Zero initialization of a value type instance is done by calling memset on this piece of memory with 0s.

- Assignment

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

- Calling the instance method

CLR supports calling instance methods on value types. This is internally done by passing the pointer to the instance as a parameter to the target method. This should sound similar to people who are familiar with C++ instance method calls, where the “this” pointer is passed as the first parameter. Since JIT owns the code generation for both the caller and callee methods, it knows how to generate correct code for a value type instance method. In other words, it expects the 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 and then passing the pointer to this copy to the target method. Consider what needs to be done at the 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 C++ syntax:

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

The stack copy is necessary for maintaining the by-value semantics so the callee only sees the copy and hence has to affect the original one.

- Passing as an argument by-reference

Passing a value type instance as a by-reference argument is just pass the pointer. Now the callee and the caller see the same instance. So any change done inside the callee will affect the caller.

- Returning a value type

Returning a value type requires the caller to provide the storage buffer. The pointer of the return storage buffer is then passed in as a hidden parameter to the callee. The callee is responsible for writing the value type into 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 code:

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

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

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

What actually happens at the call site is:

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

## Inefficiencies in the code generation with regards to value types in .NET 2.0

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 variables.
- 3) Methods with value type arguments, local variables, or return values are never inlined.

While the original intent of supporting value types in the CLR was to provide a means for creating “lightweight” objects, the actual inefficiencies in the code generation make these “lightweight” 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* in registers and avoided allocating stack space for this value type local variable altogether?

For bullet 2), the following code would mean 19 useless memcpy operations:

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

Wouldn't it be nice if the JIT could apply copy-propagation to these type local variables and morph the above code to "Console.WriteLine(point1.x + point1.y)" instead?

For bullet 3), a simple field getter of a value type turns into an explicit method call:

```
struct PointStruct() { int x; int y; public int XProp { get { return :  
    static void MyMethod3() { PointStruct point; point.x = point.y =  
        Console.WriteLine(point.XProp); } // point.XProp is a method  
        which is never inlined.
```

Currently the JIT does not perform any assertion propagation to local variables whose addresses have been taken. Common operations on value types, however, do involve taking their addresses.

### Improving value type code generation in CLR v.Next

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/ViewFeedback.aspx?FeedbackID=93858>.

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

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

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

- a) there is no operation in the current method that causes any interaction between these fields;
- and
- b) The address of this value type local variable is never exposed outside of the current method.

When the above conditions are met, the MyMethod1() listed above will be safely transformed to

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

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

And the MyMethod2() listed above will be transformed to

```
static void MyMethod2() { int x1, y1, x2, y2, ..., x20, y20; x1 =  
    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 applied to these scalars, since none of them have their address taken. As a result, the code will be reduced to

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

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

In addition, the register allocation algorithm will move the local variable into a machine register, so no stack operation will occur in MyMethod1.

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

```
static void MyMethod4() { PointStruct point; point.SomeBigMe
```

The address of *point* is taken and passed as the “this” pointer to SomeBigMethod(). What SomeBigMethod() does with this pointer is completely out of the control of MyMethod4(). In this case, *point* is not replaced by scalars. Another way to expose the address of a value type local variable is to pass it as a by-reference argument to another method. Taking the address of a value type local variable and storing it in an object 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 the following kinds of value types when 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 or object references.
- 3) The value type must be using `[StructLayout(LayoutKind.Sequential)]`, which is the default.

## Guidelines for using value types in the CLR

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

After the decision has been made to use the value type, it is time to consider the performance implications, and to determine how to help generate the best possible code. Always keep in mind that the by 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 memory operation (either operated on the stack or the heap) if this value type is not replaced by scalars.

Developers should examine the jitted code of their hot methods in a debugger to make sure the value type stack local variables are in fact moved into registers.

Try not to create value types that contain more than 4 fields. Try to create non-inlineable value type instance methods and call them on the hot path, because doing so will cause the address to be exposed. If a temporary value type instance is needed, try using value type local variables rather than the value type fields embedded in an object, because the latter are never replaced with scalars.

[1] For simplicity, let us ignore the case where a value type field is embedded in another value type.

[2] This is not true with the value type scalar replacement optimization now implemented in CLR v.Next, as described later in this document.

## 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...](#)



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 SafeA loop (which was inlined into Main by the JIT). Vance Morrison posted an 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`801501f3 8b02      mov     eax,dword ptr [rdx]
00000642`801501f5 418900      mov     dword ptr [r8],eax
00000642`801501f8 890a      mov     dword ptr [rdx],ecx
00000642`801501fa 4883c204      add    rdx,4
00000642`801501fe 493bd1      cmp    rdx,r9
00000642`80150201 7ced      j1     00000642`801501f
```

There are 7 instructions and 4 memory accesses per loop iteration, with no checks remaining inside the loop body after optimization. In this case the performance cost incurred for safety.

## Unsafe Loop:

By contrast, the unsafe version is a mess. UnsafeAccess is larger MSIL (531) because Unsafe array accesses require more MSIL instructions than safe ones. Given an array and index on the evaluation stack, safe array accesses require

single 1-byte instruction: `Idelem`. The C# compiler generates a much more complex sequence for `Unsafe` accesses:

```
IL_000c: /* 06  |          */ ldloc.0 // &array[  
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 address, there are six instructions (and bytes) required to load an array element. These extra instructions make `UnsafeAccess` larger than `SafeAccess`. When determining which methods should be inlined by the JIT, one of the most highly weighted factors is the size of the inlinee method. In this case `UnsafeAccess` was rejected for inlining, and because of this, the range check at `&intarray[0]` could not be removed. In fact the unsafe loop variant actually caused more runtime range checks to occur than the safe variant!

Finally, the presence of a pinned variable inhibits many optimizations in the JIT. As a result, the generated code for `UnsafeAccess` is far worse than the safe variant. Keep in mind that the following excerpt shows only the `UnsafeAccess` method itself, and does not even include the the loop in `Main`, as the Safe example 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]  
00000642`80150270 48b9102e352000000000 mov     rcx,20352E10h  
00000642`8015027a 488b09      mov     rcx,qword ptr [rcx]  
00000642`8015027d 488b4108      mov     rax,qword ptr [rcx]  
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]  
00000642`8015028f 4d63c8      movsxd r9,r8d
```

```
00000642`80150292 488b442420    mov     rax,qword ptr [rsp-  
00000642`80150297 468b0488    mov     r8d,dword ptr [rax-  
00000642`8015029b 4863d2    movsxd  rdx,edx  
00000642`8015029e 488b442420    mov     rax,qword ptr [rsp-  
00000642`801502a3 8b0c90    mov     ecx,dword ptr [rax-  
00000642`801502a6 488b442420    mov     rax,qword ptr [rsp-  
00000642`801502ab 42890c88    mov     dword ptr [rax+r9*.  
00000642`801502af 488b442420    mov     rax,qword ptr [rsp-  
00000642`801502b4 44890490    mov     dword ptr [rax+rdx-  
00000642`801502b8 48c744242000000000 mov     qword ptr [rsp+20]  
00000642`801502c1 4883c438    add     rsp,38h  
00000642`801502c5 f3c3    rep     ret
```

## Conclusion:

Unsafe array accesses have a lot of potential problems: correctness, heap fragmentation due to pinning, and as we have just seen, performance. I hope that this example will help developers understand that safety does not necessarily incur a runtime cost. Before attempting to evade a 'safety tax' it is a good idea to check if you are currently paying one. The first step in doing that is viewing disassembly of optimized code

-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 access](#)...

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 consequently the startup time of your NGen'ed code is better.

Framework Startup Performance Improvements: The framework is better optimized for startup. We've tweaked the framework to cover more scenarios for startup, and now layout both code & data in the framework's NGen images more optimally. What this means to you: 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 surfaces. We've also started generating stacks that are always walking 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 a system.

Better 32-bit code quality: The x86 JIT has dramatically improved [heuristics](#) that result in generally better code quality, and, in part, in 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' of the JIT, which means better null/range check elimination, as well as better constant propagation, and slight better 'smarts' in the JIT overall. What this means to you: Your managed code should run 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 run about twice as fast! In addition, we've substantially reduced the number of assemblies in which we need to regenerate NGen images. Our new Targeted Patching means that for many .NET Framework patches, we can now modify just the assemblies and not have to re-NGen any other dependent assemblies. Combined with these two features mean you and your users will spend a lot less time running your applications. Best of all, you don't have to do anything to get these benefits – they'll happen automatically when you use .NET 4.

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

If you have any comments or questions about any of these, we'd love to hear from you. Most of these features are available in the FX 4 beta 1 build.

- **NGen SxS:** Making NGen work correctly when 2 major versions of the framework are installed side-by-side
- **Multi-proc NGen:** Enabling use of multiple cores/processors during compilation to make it faster
- **Targeted Patching:** First step towards making NGen images less "frustrating" for users. It avoids having to recompile all managed assemblies when a Framework update is applied to the system.
- **No NGen in partial trust:** Deprecated support for generating and installing NGen images in partial trust applications

#### NGen SxS

To a large extent FX 4 is the first SxS release for NGen (the NGen infrastructure in .NET 1.0 and 1.1 was rudimentary). We've now architected NGen such that the `ngen.exe` tool can be used to generate NGen images for both 2.0 and 4.0 Managed assemblies without having to pass any special arguments. NGen uses the same logic as the runtime to determine which version of the assembly to load and run an application against. Thus,

`%WINDIR%\Microsoft.NET\Framework\v4.0.xxxx\ngen.exe install <4.0 assembly>` will generate an NGen image compiled against the 4.0 runtime.

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

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

`%WINDIR%\Microsoft.NET\Framework\v4.0.xxxx\ngen.exe install <2.0 assembly> /ExeConfig:<Path to a 2.0 EXE with a config file that indicates the preferred runtime>` will generate an NGen image compiled against the 2.0 runtime.

There was also work required to make the 2.0 NGen Service (`clr_optimization_v2.0.50727_32|64`) work SxS with the 4.0 NGen Service (`clr_optimization_v4.0.xxxx_32|64`). Only one service (the latest) is active at any given time – 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 parallelism is at the level of assemblies). To avoid impacting foreground activities, NGen only runs in this aggressive mode when assemblies are being compiled synchronously i.e. the NGen Service still runs on one processor. Syntactically, NGen commands (such as `ngen.exe install <assembly>`, `ngen.exe update`, `ngen.exe ExecuteQueuedItems`) will now use multiple processors/cores whenever possible. This also factor in amount of RAM when determining how many cores/processor to use. Since the parallelism is at the level of assemblies, the most effective way to use 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 (assemblies during set up using multiple cores whenever possible; other (priced) assemblies will be compiled in the background by the NGen Service at machine 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 CLR data structures” – as a consequence they’re completely fragile; any change to the underlying CLR or to any managed dependency invalidates them and requires them to be regenerated. For example, any change to the CLR or to basic assemblies like mscorlib and System that all/most assemblies depend upon invalidates all images installed on the machine. Since a machine could have several hundred NGen images, the cost of regenerating them after FX servicing events is high so 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 involve fixes to bodies of existing methods (that aren’t generic, and aren’t in across NGen image boundaries) will no longer require recompiling dependent images (the old NGen images can be used with the new serviced dependencies). Although this may sound trivial (and beg the question why the system didn’t have attribute to begin with  $\odot$ ), since NGen images had never been architected for anything but serialized JIT-ed code and CLR data, accomplishing this turned major feat. From reworking **hardbinding**, doing major performance work to the perf impact, to writing an IL post-processing tool that normalizes metadata tokens, detects Targeted Patching-compatible vs. incompatible changes, and compatible changes such that existing NGen images can be rewired to the updated dependency, plugging that tool into our build system, and revising NGen crc inlining rules, this is a major effort that several of us have been working on for months.

Some of the work for this (such as the changes to hardbinding and corresponding performance work) are included in the beta 1 build, but this feature will remain online once we start servicing FX 4. You can find out 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 <Path to assembly on intranet share>** and the generated image loaded in applications running in partial trust. We believe NGen images aren’t used in trust applications very much and our current model was broken, so we’ve disabled loading of NGen images in partial trust in FX 4. In the future (post CLR 4) we’ll rework this as part of simplifying the overall NGen story (i.e. change the most common way to generate NGen images to be issue commands from an elevated command prompt). If you’re using NGen in a partial trust application today, please let us know!

Surupa Biswas

CLR Codegen Team



search RSSing.com....

Search

# JIT ETW tracing in .NET Framework 4

May 11, 2009, 1:56 am

[» Next: Tail Call Improvements in .NET Framework 4](#)

[« Previous: Improvements to NGen in .NET Framework 4](#)



If you care about performance at a very low level, at one point you've asked yourself why the compiler, JIT, or runtime did or did not inline a certain method. Unless you worked on the compiler, JIT, or runtime, you really had no way of telling, other than trial and error (sort of like asking how a magic 8 ball comes up with its answers). We, on the JIT team, decided to fix that in CLR 4. Now you the end-user or programmer can see why the JIT, and in some cases the runtime decided to disallow inlining or tail calls. We also tell you when the JIT succeeded in inlining or tail calling a certain method. All of this wonderful information is made available by [Event Tracing for Windows](#).

First a disclaimer, I'm not an expert on ETW events, and this post isn't about ETW events in general, it is only about a couple of new events exposed by the JIT to enable performance junkies to hurt themselves. ☺

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 clevents -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 clevents -ets
```

This will create clevents.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. 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\
etw.man
```

[End Edit 1/27/2010]

To decode it further run this:

```
tracerpt cevents.etl
```

This will create 2 files: dumpfile.csv and summary.txt. The former contains the events, the latter gives a nice summary of the events. On Vista, tracerpt will generate dumpfile.xml instead of dumpfile.csv. I ran the test 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/e
<System>
    <ProviderName>Microsoft-Windows-
    DotNETRuntime <Guid>{e13c0d23-ccbc-4e12-931b-d9cc2eee.
        <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>{00000000-0000-0000-000-
        000000000000}</CorrelationActivityID>
    </System>
    <ExecutionProcessID>15476</ExecutionProcessID>
    <ThreadID>16936</ThreadID>
    <ProcessorID>1</ProcessorID>
    <Channel />
    <Computer />
    <UserData>
        <MethodJitInliningSucceededxmlns='myNs'>
            <MethodBeingCompiledNamespace>Factorial</MethodBeing
            <MethodBeingCompiledName>Main</MethodBeingCompiledName>
        </MethodJitInliningSucceeded>
    </UserData>
</Event>
```

```
<MethodBeingCompiledNameSignature>vo
System.String[])</MethodBeingCompiledNameSignature>

<InlinerNamespace>Factorial</InlinerName
<InlinerName>Main</InlinerName>

<InlinerNameSignature>void (class System
</InlinerNameSignature>

<InlineeNamespace>Factorial</InlineeNar
<InlineeName>fact</InlineeName>

<inlineeNameSignature>unsigned int32 (u
int32)</inlineeNameSignature>

<ClrInstanceId>13</ClrInstanceId>

</MethodJitInliningSucceeded>

</UserData>

<RenderingInfoCulture="en-US">

<Level>Verbose </Level>

<Opcode>JitInliningSucceeded </Opcode>

<Keywords>

<Keyword>JitTracingKeyword </Keyword>

</Keywords>

<Task>CLR Method </Task>

<Message>MethodBeingCompiledNamespace=Fa
MethodBeingCompiledName>Main;
MethodBeingCompiledNameSignature=void (class System.Str
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 of triples: **MethodBeingCompiled**, **Inliner**, and **Signature**. These are each crossed with **Namespace**, **Name**, and **Signature**. I think the name and signature are fairly obvious, but the reason they are separated instead of pretty printed into a simple element is a matter of performance. ETW is supposed to be lightweight. Computing these strings is **not** lightweight, but we felt we made a nice trade-off (just enough information to be useful, but don't spend time on making it prettier than it needs to be). **MethodBeingCompiled** is the method

VM asked the JIT to generate code for. **Inliner** refers to the method the JIT is trying to generate code for. If the **Inliner** is not the same as **MethodBeingCompiled** it is because the JIT decided to **try** and insert code for **Inliner** into the code for **MethodBeingCompiled** rather than generate a call to **Inliner**. However, just because you see a method showing up as the **Inliner**, it doesn't mean the JIT has actually succeeded in inlining it. Finally the **Inlinee** refers to the method that the JIT tries to inline (rather than generate a call to).

If this was a **MethodJitInliningFailed** event it would have 2 additional elements: **FailAlways** and **FailReason**. If **FailAlways** is true, it is a signal back to the JIT and VM that inlining will always fail for the given method regardless of the situation, and so subsequent compiles will be aborted faster (this is what **FailReason** of "It is as 'INLINE\_NEVER'" means). The **FailReason** is a free-form text field. Originally this came from some internal code we had to help us in debugging. The text is often cryptic, and is not localized. Part of the reason we did not localize it was performance. The other part is that these strings are often so cryptic many English speakers will have trouble with them. For us we often end up treating them like a GUI and search the sources for the places where that reason is returned to figure out why a particular inline failed. So if English is not your native language, don't fret too much, you're probably not missing much. This covers the 2 most common events: **MethodJitInliningSucceeded** and **MethodJitInliningFailed**.

Now we move onto the only other 2 events for this keyword:

**MethodJitTailCallSucceeded** and **MethodJitTailCallFailed**. The first two elements are the same (except replace **Inliner** with **Caller**, and **Inl** with **Callee**). The next element is the **TailPrefix** element. This element tells whether the tail IL prefix was present. For x86 this is always true because the x86 JIT does not 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 find this useful because, although some compilers expose the tail. prefix in IL, they do not have any source syntax to expose it. The programmer does not know looking only at the source what was in IL.

For **MethodJitTailCallSucceeded**, you get a **TailCallType**. This is an enum which has names that should be localized down in the **Message** element. Due to a bug in Beta 1 you will only get the numeric values. Also Vista versions of tracerpt do not do the translation from value to name. So on post Beta 1 builds with Vista or newer, you should see the pretty names otherwise expect the numbers. I will list both, especially because the number still appear in the **UserData** element. **OptimizedTailCall** means a typical tail call, where the outgoing arguments are pushed into the incoming parameter slots, the epilog is executed but instead of returning, it ends with a jump to the new method. **RecursiveLoop** means a recursive tail call, where the JIT sees that the method can't return and replaces the call with a jump back to the start of the method (skipping even the prolog and epilog). **HelperAssistedTailCall**, 2, means that the tail call cannot be accomplished directly and must go through a helper function call. These helper-assisted tail calls are slower than normal calls and only used as a last resort when the program needs to.

a tail call in order to prevent stack overflows. Thus currently you never see a [HelperAssistedTailCall](#) with [TailPrefix=false](#).

For [MethodJitTailCallFailed](#), you don't get [TailCallType](#) element, but do get [FailReason](#), which is similar to [MethodJitInliningFailed](#)'s [FailReason](#). It is also cryptic english-only text. In future posts we will explain some of the more common reasons.

So for the few people that really need to squeeze out every last bit of performance from your code, you can now see a little bit further into the JIT compiles your code. Occasionally this will enable you to inline a method in such a way as to improve inlining or tail calls. If you do remember though that **every** runtime has different heuristics and algorithms, and just because a methods was or wasn't inlined in one version doesn't mean the same will be true on a different version. 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 looking for some simple performance boosts, you can try these experiments and possibly learn something. One example we found internally moving a try/catch or try/finally up or down the call chain can often have a big performance impact because it might enable/disable inlining of a very important loop.

Grant Richins  
CLR Codegen Team  
Update 10/8/2009 - replace em-dash in logman command line with : dash that you can paste 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-tail-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 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 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 not executing the prolog or epilog multiple times. The compiler is able to treat it like a loop and hoist out loop invariants so they are only executed once instead of being executed on every iteration.

In CLR 2 the 64-bit JIT focused solely on the 'easy' cases. That means it generated code that did a tail call whenever it could because of memory benefits. However, it also meant that sometimes when it 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 performance enough to consider it an optimization, so it was only used when it 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 more often and help make F# a success. Except where explicitly called, 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 packs, QFEs, etc.) might change this in any number of ways.

The 64-bit calling convention is basically a caller-pops convention. The 'easy' cases basically boiled down to where the JIT could generate code that stored the outgoing tail call arguments in the caller's registers, 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 hard cases. The end result is that on x64 you should see shorter stacks in optimized code because the JIT generated more tail calls (worry this optimization is turned off for debug code), and function programmers (or any other compiler or language that relies on tail calls) no longer need to worry about stack overflow. The downside of course is that if you have to debug or profile optimized code, be prepared to deal with call stacks that look like they're many frames.

#### *Reducing restrictions on the easy cases*

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

- The call/callvirt/calli is followed by something other than nop instructions.

We reduced this restriction by also allowing a single pop opcode. This allowed the case where a method that returned void ended with a call to some other method that had a non-void return. The callee was only being invoked for its side-effect rather than its return value. In most cases the return value is just sitting in rax or xmm0, and does not

being left there, so in those cases, the tail call is legal and becomes an 'easy' case. However, note that having an explicit tail call, with a 'tail.' prefix is still unverifiable in this situation, so in most cases a code generator should omit the tail. prefix and just rely on the JIT optimizing the call into a tail call.

- The caller or callee returns a value type.

We changed the JIT to recognize this case as well. Specifically the calling convention dictates that for value types that don't fit in a (1,2,4, or 8 byte sized value types), that the caller should pass a 'return buffer' argument to the callee, and then copy the data from the callee's return buffer to the caller's return buffer. The CLR 2 JIT would pass a new buffer to the callee, and then copy the data from the callee's return buffer to the caller's return buffer. The CLR 4 runtime now recognizes it is safe to do so, and then just passes the caller's return buffer into the callee as its return buffer. This means even when we don't do a tail call, the code has at least one less copy, and it enables one more case to be an '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 call as having all of its arguments and the callee as having all of its arguments. Then we apply all of the other rules. It is also worth mentioning that the CLR deviates from the native X64 calling convention by adding another 'hidden' argument called the vararg cookie. It contains the GC properties of the actual call (important for the variadic parameters). The hidden argument is also counted as part of the signature for the purpose of the other rules.

- The runtime forbids the JIT to tail call. (*There are various reasons why the runtime may disallow tail calling, such as caller / callee being in different assemblies, the call going to the application's entrypoint, 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 runtime refused a tail call due to security constraints. Mainly the runtime stopped caring so much about the exact assembly instead looked at the security properties. If performing the call a tail call would not cause an important stack frame to disappear from the stack, it is now allowed. Previous to this change tail calls across or to unknown destinations (due to calli or callvirt) were rare, now they are significantly more common.

- The callee is invoked via stub dispatch (i.e., via intermediate code that's generated at runtime to 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 caller and callee. ("GC-ness" means the pointer to the beginning of an object managed by the GC or a pointer to the interior of an object managed by the GC (e.g., byref field), or 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 meet the requirements to be one of the optimized easy tail calls cases, it was turned into a regular call. This is disastrous for recursive algorithms.

many cases this prevented F# programs from running to completion without getting stack overflows. So for CLR 4 we added an alternate path that allowed tail call like properties for these specific scenarios where it was needed, but it was not easy.

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

This stack magic is not cheap. It is significantly slower than a no-op or a tail call. It also consumes a non-trivial amount of stack space. However the stack space is ‘recycled’ such that if you have a tail recursive algorithm, the first tail call that isn’t ‘easy’ will erect the TailCallHelper stack frame, and subsequent non-easy tail calls merely grow that frame to accommodate all the arguments. Hopefully the algorithm has gone through a full cycle of recursion the TailCallHelper stack frame has grown to the maximum size needed by all the n calls 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 trivial. Because of that, the JIT never uses the non-easy helper unless the user requires it with the “tail.” prefix. This was also the driving 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 feedback]

---

## Array Bounds Check Elimination in the CLR

August 13, 2009, 4:46 pm

[» Next: JIT ETW Inlining Event Fail Reasons](#)

[« Previous: Tail Call Improvements in .NET Framework 4](#)



### Introduction

One argument often made by those who dislike managed code is along the lines of “managed code can never be as fast as native



code, because managed code has to do array bounds checks.” Of course, this isn’t precisely true – it would be more accurate to say that “managed code must ensure that any indexing outside of an array’s bounds raises an appropriate exception.” If a compiler can prove statically that an array index operation is safe, it doesn’t need to generate a dynamic test.

We’re not starting from scratch here. There’s been a lot of academic (and industrial) research on bounds check elimination, and various managed code systems have 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 if the index is greater than or equal to the length. The jump target, 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 ‘greater than or equal’ 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`; 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 `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 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 common form of array accesses: accesses occur within a loop over all the indices of the loop. So, there 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, we do not eliminate the range check for the descending version of this loop:

```
static void Test_SimpleDescend(int[] a) {
    for (int i = a.Length - 1; i >= 0; i--)
        a[i] = i; // We don't get this.
```

```
a[i] = i; // We DO NOT get this.  
}
```

Some older programmers learned to write loops like this, because some early architectures (e.g., DEC PDP-8, if memory serves) the hardware addressing mode that did auto-decrement, but not auto-increment. They may have passed this habit down to middle-age programmers (of which I am one), and so on. There's also a somewhat more currently-valid argument that hardware generally supports a comparison to zero without requiring the value zero to be placed in a register. In any case, while the JIT compiler should arguably eliminate the bounds check for the descending form of the loop, we don't today know if the cost of the bounds check probably outweighs any of the other advantages. So:

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

I've put the array access on the left-hand side of an assignment in these examples, but it works independently of the context in which the array index expression appears (as long as it's 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 should know that `n` is the length of the newly allocated array in `ia`. In fact, the author of such code might think he was doing 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 system, at least today, this sort of transformation is counterproductive for the x86 JIT, since it prevents the bounds check from being eliminated. We may well extend our compiler(s) to track the equivalence of array length values 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 iteration variable if the index variable is used to index into "`a`".

The x64 JIT *does* eliminate the range checks here, by hoisting a test outside the loop, comparing `n` with `ia.Length`. If this check fails, an `IndexOutOfRangeException`. This is somewhat problematic, since without this optimization the program would execute `ia.Length` instead of the loop before throwing an exception, and strict language ser

would require those to be executed if they could possibly have an effect visible outside the method (which this example does not in fact – though proving it requires your compiler to do enough escape analysis to know that the allocated array that is written to has no effect outside the method). This semantic ambiguity is the subject of some internal debate, and we'll eventually reach consensus on how/whether to incorporate such tests in a unified JIT, or whether we need to ensure semantics, perhaps by generating multiple copies of loop bodies, as discussed below. (It's interesting to note that the hoisted test and its body would be justified by assuming the `CompilationRelaxationsAttribute` defined in section I.12.6.4 of the ECMA CLI specification for boundary error exceptions everywhere – whereas the specification requires this given 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 customer code whose correctness is affected by this issue.

### Redundant array accesses

OK, while we're slightly embarrassed by the previous “multiple non-local to the length” case, let's cheer ourselves up with something we do well: We're pretty good at eliminating redundant bounds checks. In this 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, the x86 JIT treats it as a common subexpression, so the first result is re-used. And this works not just within “basic block boundaries”, but can work across control flow, as demonstrated by:

```
static void Test_RedundantExBB(int[] a, int i, bool b)

    k = a[i];
    if (b) {
        k = k + a[i];
    } else {
        k = k - a[i];
    }
}
```

As before, the first “`a[i]`” gets a bounds check, but the two subsequent occurrences of “`a[i]`” do not. The x86 JIT also treats the expression as a common subexpression, re-using the result from the first read of “`a[i]`”.

It is not the case that bounds check elimination *only* works in the presence of common subexpressions; it also works when the result is a common subexpression. Consider this variation on the first case:

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

The JIT compiler obviously can't tell whether "i" and "j" will have same value at runtime. But it can tell that they *might*, and that if the "a[i]" on the last line will return the value written there on that line. So we cannot treat the "a[i]" expressions on the first and last line of the body as common subexpressions. But the assignment on the second line can't affect the *length* of the array "a," so in fact the check for the first line "covers" the "a[i]" on the third line – the 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 enumerate over the elements of an array using C# `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 here: of the cases listed, this one is the only one that's sensitive to whether the original source program (C#, in this case) was compiled to IL with the `/optimize` flag. The default for `csc`, the C# compiler, is not to optimize, and in this mode it produces somewhat verbose IL for the range check that doesn't fit the pattern that the JIT compiler looks for. So:

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

### Arrays in global locations; concurrency

Here's a case where we don't eliminate the bounds check, but we aren't too embarrassed by 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 `Test_SimpleAscend`. The difference is that `Test_SimpleAscend`'s array argument, whereas `Test_ArrayInGlobal`'s array is accessed via a static variable, accessible to other threads. This makes static elimination of the bounds check for “`v[i]`” at the very least dicey. Let's say we have `v` and that “`v`” initially holds an array of length 100. On the iteration where “`i`” reaches (say) 80, we check “`i < v.Length`”, and it's still true. Now another thread sets “`v`” to an array whose length is only 50. If we proceed ahead with the array store without a dynamic check, we're writing past the end of the array – type-safety and security are lost, game over. (Obviously, the same reasoning would apply for an array held in a location accessible to multiple threads – an object field, element of another array, anything not local to the running thread.)

So we don't do this, for good and solid reasons. If we cared enough about type-safety, there is a technique that would allow us to eliminate these bounds checks. But it would require us to couple otherwise-unrelated optimizations. For example, if happens, the code for accessing a static variable in the presence of multiple threads can be moderately costly, so it's good to treat those as candidates for subexpression elimination. The x86 JIT does in this case (the x64 JIT does not). So the optimizer in essence synthesizes a local variable to hold the value of the array. If we do this, then we are back in the `Test_SimpleAscend` situation, and the bounds-check elimination is legal. But doing this requires that the static variable be reanalyzed and converted into a local. So it's at least a bit complicated.

### Parallel arrays

Next we consider a case that involves what are sometimes called “parallel arrays” (in the sense of their structure, not in the sense that they will be used by multiple threads):

```
static int Test_TwoArrays(int[] ia1, int[] ia2) {  
  
    // The programmer knows a precondition: ia1.Length  
    == ia2.Length  
  
    int sum = 0;  
  
    for (int i = 0; i < ia1.Length; i++) {  
  
        // Below we eliminate the ia1 check, but  
        // not the ia2 check.  
  
        sum += (ia1[i] + ia2[i]);  
    }  
  
    return sum;  
}
```

Much as with `Test_ArrayCopy1`, the x64 JIT hoists a test comparing `ia2.Length` and `ia1.Length`, immediately throwing the bounds-check exception if the test fails. If the test succeeds, range checks for

array accesses in the loop are eliminated. The same comments about semantic issues with such a test apply. The x86 JIT takes a more approach: it does not hoist a test, so it only eliminates the bound for the access to the array `ia1` whose `Length` bounds the index variable.

We could resolve the two approaches. The mechanisms proposed sort of problem in the research literature have the common property they require, at least in some cases, generating code for the loop times, under different assumptions, and synthesizing some sort of test to determine which version of the loop should be executed – this is essentially the test that the x64 JIT is already creating. Generally, check exceptions are rare – if the programmer wrote the code assuming or she had some reason to believe that the index expression “`ia2.Length >= ia1.Length`” was safe. So we could synthesize a test on that basis. In our case assuming the compiler proved that neither argument variable “`ia1`” or “`ia2`” was modified in the loop, then a test “`ia2.Length >= ia1.Length`” (the one the x64 JIT generates) outside the loop would allow us to execute an optimized version of the loop, with no bounds checks for either array access. If this test failed, however, we’d need to execute an unoptimized version of the loop to be completely semantically correct. You’d have to evaluate this test carefully, since it’s code that doesn’t appear in the original program. In particular in this case, you’d have to worry about whether either of “`ia1`” or “`ia2`” were null. If they are, you want to ensure a pointer exception to occur at the appropriate point in execution, rather than some code the compiler made up. So the synthesized test would 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 loop – it throws the exception “early” in the case. Under the “purist” viewpoint, this is incorrect because if the test fails, the semantics require the program to execute some number of iterations before throwing the exception, and those iterations might have side effects. In many cases, we might be able to prove that the loop does not have side effects, and therefore use the x64 JIT’s strategy of semantic blessing. For example, a loop that computed the sum of all elements in a local variable would have side-effect only that local variable. If the exception causes control flow to leave the method, the value of the 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 pointer of both arrays as a separate argument, and used that as the loop variable. We could do something similar, synthesizing a test of that loop based on the lengths of both array pointers.

### ***Explicit Assertions***

Another suggestion that has been made is to allow the programmer to provide the relevant test in the form of a contract assertion (of a guard condition). This would essentially provide semantic “permission” to execute the loop immediately if the test is violated, avoiding the need to have an unoptimized version of the loop. There are many things to be said in favor of this sort of proposal: they can allow bounds checks to be eliminated in situations more complicated than those for which the compiler can easily infer a test, and the invariants they express are often useful for program documentation as well. Still, I also worry somewhat about this proposal. In many common cases, it’s easy enough to infer the invariant, so we should avoid requiring the programmer to add assertions in those cases. More importantly, if the programmer adds an assertion ex-

it to eliminate a bounds check, how does the tool chain indicate whether he or she has been successful? And, if not, why not? These sort of issues merit some more thought.

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

The right strategy in this area is obviously a little muddled. Consider feedback is welcome!

### Copy loop

Here's another example, somewhat similar to the Test\_TwoArrays:

```
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];  
  
    }  
  
    return res;  
}
```

As you might expect from previous examples, since we use the length of “res” as the loop bound, we eliminate the bounds check for the access to “res”. But we do not eliminate the check for the access to “ia1”. In Test\_ArrayCopy1, to eliminate this we'd need to do a better job of tracking equivalences of array lengths with local variables or other lengths. We don't do this today, but it's certainly a plausible enhancement we might do. The x86 JIT leaves the bounds check for the access “ia1[i]”, while the x64 JIT hoists a bounds-check out of the loop, as discussed above (and with the same difficulties discussed above).

While it would be nice for us to eliminate the bounds checks case, this, if you're copying arrays, there are many reasons to use the `Array.Copy` routine rather than writing an explicit copy loop like this appear in these examples:

- **Advice 4:** when you're copying medium-to-large arrays, use `Array.Copy`, rather than explicit copy loops. First, all your bounds checks will be “hoisted” to a single check outside the loop. If the arrays contain object references, you will also get efficiency “hoisting” of two more expenses related to storing into arrays of object types: the per-element “store checks” related to array covariance can often be eliminated by a check on the dynamic types of the arrays, and garbage-collection-related writes will be aggregated and become much more efficient. Finally,

will be able to use more efficient “memcpy”-style copy loops in the coming multicore world, perhaps even employ parallelism if the arrays are big enough!)

### Multi-dimensional Arrays

The CLR, and C#, support real multi-dimensional arrays – in contrast to C++ or Java, which (directly) support only one-dimensional arrays. To represent two-dimensional arrays, you have to simulate them, either through classes that represent the 2-d array as a large 1-d array, and do appropriate index arithmetic, or as an “array-of-arrays.” In the latter case, even if they are allocated originally to form a “rectangular” array, it’s hard for a compiler to prove that the array *stays* rectangular. 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 (just as the length of a regular 1-d array is). This makes removing bounds checks in each dimension more tractable. A notable advantage is that indexing calculations become easier when the array is known to be “rectangular.” (With a good optimizer and appropriate aggressive inlining, C++ template-class-based simulations of multidimensional arrays can get similar indexing calculation code.)

Unfortunately, we aren’t yet able to remove any range checks for accesses in multi-dimensional 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 clearly need to eliminate bounds checks for multidimensional array accesses if we want to generate reasonable code, say, a matrix multiplication.

- **Advice 5:** Until we get this right, I would suggest that .NET do what many C++ numerical programmers do: write a class to implement your n-dimensional array. This would be represented as a 1-dimensional array, and the relevant accessors would convert n indices into 1 via appropriate multiplications. Very likely, this almost certainly wouldn’t eliminate the bounds check into the 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 cases, and the costs of bounds checks usually

that great when we don't eliminate them. And, as I mentioned at beginning, we have to keep in mind that the compiler we're talking about is a dynamic JIT compiler, so we must carefully balance adding extra optimization that slows the compiler against the gains of that optimization. Still, if we don't eliminate a bounds check that we have in a small, tight loop that's important to the performance of program, I doubt you'll find these excuses very satisfying. I hope this post convinces you that we're well aware of the problems. The fix almost certainly holds some mechanism for applying extra compilation effort to methods whose performance matters a lot, either by doing extra work in some form of 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 elimination will 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 possible strings that might show up in the FailReason field of the MethodJitInliningFailed 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:

- "Inlinee is NoMetadata" - this means the inlinee is not normal managed code. The most common case is [dynamic methods](#). Another possibility is one of the many kinds of stubs the CLR uses internally for things like PInvoke marshalling, delegates, remoting, etc.
- "Inlinee is debuggable" - the inlinee was compiled or loaded in such a way that it told the runtime not to optimize its code (C#'s /O- switch is one example).
- "Profiler disabled inlining globally" - a profiler passed [COR\\_PRF\\_DISABLE\\_INLINING](#).
- "Profiler disabled inlining locally" - a profiler passed [COR\\_PRF\\_MONITOR\\_JIT\\_COMPILATION](#) and then set [\\*pfShouldInline](#) to FALSE in [ICorProfilerCallback::JITInlining](#).
- "Inlinee is not verifiable" - the inlinee does not have [SkipVerification](#) permission and is not verifiable. This is done so that verification exceptions are easier to debug.
- "Inlinee is marked as no inline" - the inlinee is explicitly marked with [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 implementation limitation such that it can't find the right method for the JIT to inline (but don't worry, it still finds the right one to call). `IMetadataEmit::DefineMethodImpl` for how this is done at the IL level. My understanding that the C# language does not allow this, but it does.
- "Targeted Patching - Method lacks `TargetedPatchingOptOutAttribute`" this relates to a new feature in CLR 4 where NGEN images are version resilient. In a nutshell, we hope to be able to apply a patch to mscorlib.dll and **not** have to recompile all the other native images in the machine that depend upon it. This should only apply to methods in the .NET framework class libraries because they are the only ones that can **opt into this feature**. For more information, see this pre-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 have those 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 poor warm startup due to JIT-ing of method calls OR you are working with a library that gets loaded into several processes on a machine and you would like to reduce the library's impact on the total working set of the machine.)

For the purposes of this walk-through, I assume you already have a Visual Studio solution for your application. This series of steps will provide guidance on how to invoke NGen for your application from within the Visual Studio solution. Using NGen via the set of steps outlined below will result in native images created only on the specific machine where the steps are run. Note also that this article does not provide instructions on how to invoke NGen in an installer package; a future article will talk about that scenario.

## Adding a post-build event to run NGen.exe

Setting this up involves a fairly straightforward set of steps.

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 Events...** window by specifying each command on a separate line.

Under the **Project** menu go to **Properties** and in the **Debug** tab, click 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, an 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 use NGen.exe 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 via the **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 DLLs.

#### Further Reading

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

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

- 1> Visual Studio is not being run with Administrator credentials. NGen is an Admin tool and the post-build step will fail with the following error if 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 these tasks.
```

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

### **Ensuring native images got loaded during runtime**

A native image file has a “.ni.dll” or “.ni.exe” file extension. Note that you don't need to do anything special to run a native image (like providing a path to the native image file). When you run hello.exe (the IL assembly), Common Language Runtime's assembly loader will check if a native image file exists for this IL assembly and will just load that instead of the IL assembly.

The first thing to check for is that the native image files were loaded during runtime. The next thing to check for is whether the JIT got used to compile some methods in an assembly even though the native image for that assembly got loaded.

There are a couple different ways to validate that the native image was loaded during runtime.

- 1> Use the SDK tool Fusion Log Viewer: Easiest if running outside of Visual Studio

Launch the tool via the Visual Studio Command Prompt. Under **Fusion Log Viewer > Settings**, enable **Log All Binds to Disk**. (You may also need to change **Enable custom log path** and provide a custom log path).

Run your application.

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

The format of each entry in this log is such that Application column contains the name of the executable 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 “WF matching native image found.” If the native image for that assembly did not load, there will be no log entry.

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

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

While debugging, from the **Debug** menu go to **Windows** and **Modules**, to see a list of the 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](http://blogs.msdn.com/jmstall/archive/2006/10/04/debugging_5F00).

#### **Ensuring that the JIT did not get used during runtime**

You might be wondering what the difference between the previous section “Ensuring native images got loaded during runtime” and this one is. If a native image file got loaded for an assembly, the JIT should not fire right? Well, not quite. Sometimes, even though the native image might have been used for the bulk of the method in that assembly, the JIT does load some methods, or types of code within the assembly being executed; for example, some generic instantiations, dynamically generated code, multiple AppDomain scenarios etc. So, even though the native image file might have been loaded for an assembly and we may have used native code from that assembly for most of our method execution needs, some methods, we may have to fall back to JIT. It is important to know which methods are for your application.

Moreover, if your application has several DLLs, it may not be a trivial task to ensure that each DLL had a corresponding “.ni.dll” loaded by inspecting either Fusion Log Viewer content or the Modules window content. Using the JitCompilationStart MDA will help in such scenarios well.

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

Now let's look at the practical steps for how to set up your Visual Studio configuration correctly so that 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](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 solution with the contents below and set the **Properties** set **Copy to Output Directory** to **Copy Always**.

```
<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 **Project** menu go to **Properties** and in the **Debug** tab the check box for **Enable Unmanaged Code Debugging**.

5> Finally, in order to enable the MDA infrastructure to look at the hello.exe.mda.config file you created in Step 2, you need to update registry. Under **HKEY\_LOCAL\_MACHINE\Software\Microsoft\NetFramework**, create a F sub-key called "MDA" and set its value to "1".

Yes I understand this is an unnecessarily large set of steps to go through to enable this particular MDA, we hope to improve this experience in a future release of Visual Studio.

Now that the JitCompilationStart MDA is enabled, all you need to do is if a method does get JIT-ed, an unhandled exception dialog box will be generated of the type below.



Select Break in the dialog box to see the line of disassembly for which the JIT got loaded. The Call Stack window will also show the unmanaged stack at which the break point occurs. This will provide a clue as to what 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](http://msdn.microsoft.com/en-us/library/d21c150d(VS.80).aspx)

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

#### **Wrapping it up!**

This article summarizes the very basics of using Visual Studio to run NGen for your application. This article hopefully articulates all the various steps needed to get the environment right, and points out potential pitfalls along the way. We'd love to hear what you think! Have you used Visual Studio in the past to enable NGen for your application? If not, what do you use?

was your experience? Please use the comments section below for 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 loaded have native images and the second one will involve the same application where most of the managed assemblies will NOT have native images.

#### Further Reading

1> Performance Benefits of NGen: <http://msdn.microsoft.com/en-us/magazine/cc163610.aspx>

2> A model for cold startup time:  
<http://blogs.msdn.com/vancem/archive/2007/04/09/a-model-for-cold-startup-time-of-an-application-on-windows.aspx>

#### **Getting Started with Xperf**

Download Windows SDK: <http://msdn.microsoft.com/en-us/performance/cc825801.aspx>

Xperf is an ETW based performance measurement tool for Windows available publicly. This tool enables drill down analysis into how much machine-wide CPU time was spent in modules of an application.

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

#### Further Reading

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

2> Additional Information on Xperf:  
<http://blogs.msdn.com/pigscanfly/archive/2008/02/09/xperf-a-new-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 known large application that has components written in managed code and have native images; Visual Studio 2010. The scenario will launch a HelloWorld WPF application in Visual Studio 2010, wait till the UI is responsive, and shutdown. In order to collect CPU time traces for later analysis, Xperf will be used to enable tracing 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  
Flag)
```

2> Run scenario of interest

```
"C:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE\dev  
"c:\users\xyz\documents\visual studio 2010\Projects\WpfApplication1\Wp
```

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

```
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. This will open and display a graphical view of the data in the file, similar to the one shown below.



*Screenshot 1: Windows Performance Analyzer*

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

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

Expanding the devenv.exe process displays a view of the percentage of time spent in each module loaded within this application. Notice in Screenshot 2 below that the bulk of the time was spent in clr.dll (20.2% total across the machine, which means that it was ~30% of the time within the process devenv.exe) and clrijit.dll (8.68% total across the machine, which means that it was ~12.6% within the process). Note that some time was spent in the JIT even though most of this scenario was running native images. The use of native images can be seen in the fact that native files are listed in the module list (mscorlib.ni.dll, system.xaml.ni.dll etc).



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

#### Further Reading

- 1> Detailed guidance on using Performance Analyzer: All F -> Windows Performance Toolkit -> Windows Performance

Help. Select Quick Start Guide: WPF Basics -> Detailed Walkthrough.

### **Warm Startup Time without Native Images**

Now let's turn our attention to what the startup time characteristics look like when native images are not used in the Visual Studio 2010 application. Solely for purposes of this demo, the numbers below were generated by uninstalling all native images for Visual Studio and .Net Framework assemblies. It is NOT recommended that you uninstall native images in general. Executing the sequence of actions used previously, to generate traces again for this scenario, we get the graphs seen below.

The increase in the total CPU utilization by devenv.exe (49.34%) compared to the analysis a tad bit, but we can see that the time spent in clr.dll went up to 20.5% total, which means ~41.5% of the time spent in devenv.exe was spent in clr.dll. This is a large increase from the previous scenario where only 12.6% of the time spent in devenv.exe was in clr.dll. Furthermore, the time spent in clrjit.dll went up to 16.03% total, which means ~32.5% of the time spent in devenv.exe was spent in clrjit.dll. This is also an increase from the previous scenario where only 12.6% of the time was spent in clrjit.dll, almost double! Also note that the list of modules loaded no longer shows modules that end with a .ni.dll file extension, indicating that native images were not loaded.

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



*Screenshot 3: Windows Performance Analyzer, CPU Utilization Sun Table drilldown without native images present*

### **Wrapping it up!**

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

Lakshan Fernando, Pracheeti Nagarkar

CLR CodeGen team

search RSSing.com....

Search

# NGen: Measuring Working Set with VMMap

April 27, 2010, 10:59 am

[» Next: NGen: Creating Setup Projects](#)

[« Previous: NGen: Measuring Warm Startup Performance with ...](#)



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>”.



Screenshot 1: Overall View

In Screenshot 1 above, observe the various memory types listed in the rows in the upper pane (Image, Private, Shareable etc.). For each memory type (for example, Private), there are several columns that detail how that memory type breaks down; how much is the Committed memory, Total Working Set, Private Working Set, Shareable Working Set etc. Note that the precise definition of each of these is available from the Help->Quick Help menu in the UI. The

Image memory type is easier to use for tracking purposes because it allows us to see the executable file that launched the current process being analyzed. This also enables a drill down into which other files were loaded into the process and how much of memory they consume. The goal here is to reduce the Total WS (benefits the application) and reduce the Private WS (coupling with other processes and thus increasing machine wide resource usage).

In order to see the detailed break-down of the Image memory type, click on the row named Image in the upper pane, select the row named Image. The lower pane will now show the list of all files loaded into this process and the resource usage for each file. See Screenshot 2 below.



Screenshot 2: Image memory type view

For each file (file name is visible in the right-most Details column, see Screenshot 2 above), the resource usage of the file is further broken down into Size, Committed, Total WS, Private WS, Shareable WS and Shared. We are interested in the working set columns (Total, Private, Shareable, Shared).

Let's understand one of the rows shown above, for the file mscorelib.dll:

Size = 13,936K. This refers to the size of the image which comes from the PE header of the file. For the curious, to see where this number comes from, do the following: From a Visual Studio SDK command prompt use "ildasm /dump /headers <PathToFile>", and look under the Optional Headers section for "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 shared 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 shared 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 being shared with another process that also needs this file. If there were no other process that needs this file, this number could be zero.

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

#### Further Reading

1> VMMap: <http://technet.microsoft.com/en-us/sysinternals/dd535>

#### **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 that only essential modules get loaded into the process.

If the application and its entire closure of dependencies has been loaded, the general case is expected that the JIT will not get loaded. In the column in the lower pane of VMMap, check that clrjit.dll (.Net Framework v4.0) or mscojit.dll (Pre-.Net Framework v4.0) has not been loaded. If it has been loaded, it indicates that something within your application is generating native code. This may be an entire dependency that was not NGen'd or it may be a few methods within an assembly for which native code could not be generated. For the former case, Fusion Log Viewer can be used to determine the assembly for which a native image could not be found. In the latter case, the JIT Managed Debugging Assistant helps answer the question around which method the JIT is getting invoked for. See the section 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 WS. This does not seem like a whole lot when loaded independently. However, in order to invoke the JIT, the CLR's engine executes more code and as a result more pages from the engine's DLL (called clr.dll in .Net Framework 4.0 or called mscorwks.dll in prior releases) are pulled into the Total WS. In the example I used, the Total WS of my application went up by ~100K. That's not all. Since a particular assembly say A (part of it) needed to get JIT compiled, the IL for that assembly also needs to be loaded. Depending on the size of this assembly (or the size of the method that needed to be JIT compiled), the IL also contributes to the Total WS. Moreover, since the JIT compiled code cannot be shared across process boundaries, if A.dll was also needed subsequently by another process on the machine, that new process would need to incur its own WS cost to JIT compile the code.

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

#### 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](http://msdn.microsoft.com/en-us/library/fw872k46(VS.80).aspx)

#### **The Basics: Using the GAC**

In some cases, even when the JIT does not appear in the list of modules loaded in the process, it is possible that both the IL as well as the native image will get loaded for a particular assembly. Note that the application main executable will be listed twice in VMMap – the first listing corresponds to mapping the IL and the second corresponds to loading the native image. This is expected. See Screenshot 3 below. Note that when a native image is loaded for an assembly, the Details column will show a path of the form C:\Windows\assembly\NativeImages\_\*.

#### Vmmap\_Screenshot3

Screenshot 3: Example of both IL (foo.dll) and native image (foo.ni.dll) contributing to Total WS

However, for any other managed assembly in the list of modules in the process, having both the IL and the native image contribute to Total WS is preventable. A couple of the main reasons why both IL and native image may appear loaded are if the assembly is a mixed-mode assembly (the assembly being loaded lives outside the GAC). For the former case, loading both IL and Native Image for a mixed mode assembly is not preventable. In the latter case, installing the assembly into the GAC will stop the additional load of the IL. Again, it is important to keep in mind that the additional load of the IL assembly depends on the size of the assembly – it may not be large enough to make you want to install the assembly into the GAC.

Aside: Speaking of the GAC, prior to .Net Framework 3.5 SP1, it was strongly recommended that all strong name signed libraries used by an application be installed into the GAC. The reason was that 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 the assembly's signature before permitting the load of the native image for security reasons. Starting with .Net Framework 3.5 SP1, a feature called strong name bypass eliminates the need to compute this hash for most cases. In Screenshot 3, the assembly foo.dll is strong name signed but lives outside the GAC. The IL still gets mapped and contributes to Total Working Set because we do not incur an additional working set hit from clr.dll for computing the strong name hash. As an experiment, if the strong name bypass feature is disabled, the 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-Vis**

When rebasing occurs, the Private WS increases and the Shareable WS decreases; this is not optimal 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 the NativeImageBaseHeader. From a Visual Studio Command Prompt, do a "link.exe /dump /headers <PathToNativeImageFile>". Screenshot 4 below shows this.



*Screenshot 4: Preferred Base Address*

So who chooses the base address? Unless you specify a base address during compile time (for instance, via the /baseaddress option to `cc`) the CLR will pick one! If a base address was not specified during compilation, the compiler assigns a default base address of 0x4000000. When creating a native image, the CLR translates this default address for the executable to 0x30000000 and for a DLL to 0x31000000. Screenshot 4 shows the DLL default above, as picked by the CLR. This address is used by the operating system to load the PE file at. VMMap shows the address was used in the very first column "Address" in the lower pane.

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



*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 that are references to items (like strings) within the native image itself. If the native image gets loaded at an address other than the preferred base address, the CLR needs to adjust those references (known as performing fixups) – this is done by writing to the page that contains the reference, thereby creating private pages that cannot be shared with other processes that might also want to load the same native image. Using VMMap, when rebasing occurs, the Private WS number increases and the Shareable WS decreases. This is not optimal for machine wide memory usage.

The use of hard binding further complicates this analysis. Note that managed applications utilized hard binding for native images, when the application launches, the hard bound dependencies get loaded first. In case any of the hard bound dependencies gets rebased, the dependency itself will need to have fix ups, creating private pages. However, the application's native image that stores references to the dependency native image will need fixups as well. The application's WS will increase for two reasons – once for the rebased dependency, 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 WS – 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.aspx#S5>

### **Impact of Base Address Collisions (Rebasing): What about Vista?**

Starting with .Net Framework 3.5 SP1, on Vista machines (and newer Windows operating systems), native images are opted into an OS feature called ASLR (Address Space Layout Randomization). This means the preferred base address in the PE file of a native image, and loads it at a random base address. As a result, it does not matter what preferred base address was assigned to a library; it will not be used.

As mentioned above, if your application's distribution is restricted to newer operating systems, base address collisions is a class of problem that goes away.

#### Further Reading

1> Address Space Layout Randomization: <http://technet.microsoft.com/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 managed processes on a machine. This is particularly helpful for server scenarios where there may be multiple managed applications running on the machine, that depend on a common set of managed libraries. If an application has already loaded a native image, and a second application is started that uses the same native images, the pages from the native image that are still marked shareable, will be shared. 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 image for foo.dll, the "Shareable WS" is listed and the "Shared WS" is nothing. This indicates that an 8K chunk of memory in the native image for Dep1.dll is available for sharing, in case another process wanted to share it. When we launch a second application that uses Dep1.dll, hit the Refresh option in VMMap to show the new working set numbers. We'll now see that the "Shared WS" number is 8K. This indicates that of the memory from Dep1.dll that was available for sharing, it has been shared all of it.

### **Wrapping it up!**

This article summarizes the very basics of using VMMap to analyze the impact of NGen on the working set of a managed application. This article hopefully articulates how VMMap can be used and points out what to watch for in the measurements. We'd love to hear what you think! Have you used

VMMMap in the past to analyze the impact of NGen? If not, what do you think about it? How was your experience? Please use the comments section below to share your 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 VMMMap](#)

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](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 NGen Custom Action in Visual Studio Deployment Project  
<http://msdn.microsoft.com/en-us/library/3hwzzhyd.aspx>

### Getting Started with WiX

Assuming you have Visual Studio 2010 installed, the next step is to work against WiX 3.5 from <http://wix.sourceforge.net/releases/3.5.1419.0>.

This version is officially the one used for Visual Studio 2010 RC, but work against Visual Studio 2010 RTM as well. The version of WiX that's supported on Visual Studio 2010 RTM is scheduled to release soon.

The walk through below assumes that a Visual Studio project for a Windows Forms application called *HelloWinForm* exists already. To create the WiX project that will install this application, in your Visual Studio in **Solution Explorer** right click on the solution, **Add a New Project** under 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 **Add Reference**. In the **Projects** tab, select the *HelloWinForm*. This will ensure that the *HelloWinForm* project is built prior to the *WixProject1*, and the input to *WixProject1* is the output of the *HelloWinForm* project.

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

```
<File Id="WindowsFormsApplication1File" Name="$(var>HelloWinForm)
```

### 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 Installer XML Toolset 3.5 -> WiX Documentation -> Using WiX in Visual Studio -> Creating a Simple Setup.

### **Adding NGen steps to the WiX Project**

Now that we have a basic WiX project working without NGen, the next step is to incorporate NGen into 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 **Browse** tab, find **WixNetFxExtension.dll** under the **Windows** folder.

directory (the full path will likely be something like C:\Program Files\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 below.

Before: `<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi" ...>`

After: `<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi" xmlns:netfx="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* project. To do that, add the following line under the `File` element.

```
<netfx:NativeImage Id="WindowsFormsApplication1File.exe" Platform="AnyCPU" />
```

Detailed documentation for the WiX schemas can be found via the 'Help' menu, All Programs -> Windows Installer XML Toolset 3.5 -> Windows Installer XML (WiX) Help -> WiX Schema References -> NetFx Schema -> NativeImage Element (NetFx Extension).

The resulting WXS file will look as follows –

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi" xmlns:netfx="http://schemas.microsoft.com/wix/NetFxExtension">

  <Product Id="3645a635-5cb6-48da-8a2a-6d9bd9778ae" Name="Wix35Project1" Language="1033" Version="1.0" b331-4ca3-94b1-2bfd652a280a">

    <PackageInstallerVersion="200" Compressed="yes" />
    <MediaId="1" Cabinet="media1.cab" EmbedCab="yes" />

    <Directory Id="TARGETDIR" Name="SourceDir">
      <Directory Id="ProgramFilesFolder">

        <Directory Id="INSTALLLOCATION" Name="Wix35Project1">
          <Component Id="ProductComponent" Guid="13b3ea2'4e39-b225-62b67d2255d0">

            <File Id="WindowsFormsApplication1File" Name="$(var.HelloInfo)</File>
            <netfx:NativeImage Id="WindowsFormsApplication1File.exe" Platform="AnyCPU" />
          </Component>
        </Directory>
      </Directory>
    </Directory>
  </Product>
</Wix>
```

```
</Directory>  
</Directory>  
  
<FeatureId="ProductFeature" Title="Wix35Project1" Level="1">  
    <ComponentRefId="ProductComponent" />  
</Feature>  
</Product>  
</Wix>
```

Specifying a priority of 0 ensures that the native image compilation done synchronously when the installer is running. Specifying a priority of 1, 2 or 3 will make the compilation occur in the background. Prior to the effect that assemblies are immediately compiled in the background priority 2 means the assemblies will be compiled after the priority 1 assemblies are done, and priority 3 assemblies are compiled at maximum time. If no priority value is specified, the priority will default to be 3. The toolset will issue an "ngen install" or "ngen uninstall" action depending whether the resulting MSI is being used to install the application or to remove it from the machine. The toolset will always issue an "ngen update /quiet" command, in order to trigger the NGen service to recompile 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-an-msi-based-setup-package-using-wix.aspx>
- 2> Introducing NGen Support in WiX:  
<http://installing.blogspot.com/2006/06/ngen-support-in-wix.html>
- 3> Synchronous versus Asynchronous Native Image compilation:  
[http://msdn.microsoft.com/en-us/library/ms165074\(v=VS.80\).aspx](http://msdn.microsoft.com/en-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 to invoke NGen.exe in an installer for your application. This article articulates the various steps needed to get the environment setup, points out nuances along the way. We'd love to hear what you think you used WiX in the past to enable NGen in your installer? If not, what technology do you use? How was 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 indirect calls), the callee must have the same permissions as the caller. If you want to do a tail call, use a homogenous appdomain, grant the caller Full Trust, or put the caller and the callee in the same 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 generate a tail call from a method marked as the [entrypoint](#) for a module. The idea is that programmers like to see their Main method at the bottom of the stack always. There is no way around this restriction.
- "Caller is marked as no inline" - If there is no "tail." instruction prefix and the caller is explicitly marked with [MethodImplOptions.NoInlining](#), then the VM assumes the programmer really wants that method frame to remain on the stack and not get elided via inlining or tail calls, and so it prevents tail calls from that method. If you want to do a tail call, either explicitly add the "tail." prefix to the call or remove the NoInlining flag.
- "Callee might have a StackCrawlMark.LookForMyCaller" - certain methods in mscorelib rely on a stack walk to determine their caller. They are marked to prevent inlining and also to prevent direct tail calls. This will only happen if the callee is known, and is inside mscorelib. There is no way to generate tail calls directly to these methods.

- "Caller is a CER root" - See the [Constrained Execution Regions](#) MSDN.

From x86 JIT, we get this list of failure reasons (again in no particular order):

- "Caller is synchronized" - The caller is marked with [MethodImplOptions.Synchronized](#). The JIT needs to leave the current frame on the stack until after the callee finishes in order to know when to release the runtime-implemented locking.
- "Caller is varargs" - This is just an implementation limitation of the JIT. For more information about varargs in C# (**not** the params keyword), search for [\\_\\_arglist](#).
- "Caller requires a security check." - The caller is marked with [mdRequireSecObject](#) for [imperative 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 call, 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 call" - Same as above.
- "Return types don't match" - The caller and callee must have the same return type. If you want to do a tail call, change the return type to match.
- "Localalloc used" - This is just an implementation limitation of the JIT. In C# if you use [stackalloc](#) then the JIT cannot be sure of the intended lifetime, and so it goes safe and prevents the tail call. If 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 allocate a buffer. Normally the JIT reuses the return buffer for the caller to avoid a copy, but sometimes it can't because it now has to do a copy after the callee returns, it can't do a tail call. If you want to do a tail call, use an out parameter rather than returning the value.
- "Changed into handle" - The C# expression `'typeof(XXX).TypeHandle'` involves a call to the property method [get\\_TypeHandle](#). The JIT embeds that whole expression (including the call) into a simple embedded constant (the TypeHandle as provided by the VM). We think that's better 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 regular functions with try/catch/finally clauses, unless the call uses the "tail." instruction prefix. If you want to do a tail call remove the exception handling or add a "tail." prefix.
- "found symbol with address taken" - if the call doesn't use the "tail." instruction prefix and the method takes the address of a local, the JIT doesn't do enough analysis to see if it is address-escaped (meaning the callee uses the address to access the caller's local) and so it just tries 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 "[requires a security check](#)".
- "caller's declarative security" - This is the same as the VM's "[Caller is declarative security](#)".
- "not optimizing" - The JIT disabled all optimizations, and so it only performs a tail call if the "tail." prefix is present. If you want to do a tail call, either add the "tail." prefix or re-enable optimizations. Some

reasons why JIT optimizations might be disabled include: using [MethodImplOptions.NoOptimization](#), a method that is too big or too complex to optimize, running under a debugger, and certain compiler switches.

- "localalloc" - This is the same as the x86 JIT's "[Localloc used](#)", except the 64-bit JIT will do a tail call if the call explicitly uses the "tail." instruction prefix.
- "GS" - The method uses local buffers (unmanaged arrays) and the JIT adds extra code to detect buffer overruns before they can be executed. These extra checks are incompatible with tail calls in our current implementation. The name comes from the [C++ compiler's /GS command-line switch](#), and attempts to prevent many similar issues that appear in unsafe managed code.
- "turned into intrinsic" - The 64-bit JIT cannot tail call certain methods effectively turn into special code. This is similar to the x86 JIT's "[Turned 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 that don't require any conversion at the hardware level). This is similar to the x86 JIT's "[Return types don't match](#)", but the 64-bit JIT is slightly more permissive.
- "processor specific reasons" - The caller and callee's signature is different enough that the calling convention makes it hard (or impossible) to do a traditional optimized tail call. This is usually caused by the caller having more (or bigger) arguments than the callee. On x64 if the instruction prefix is used, the JIT will generate a HelperAssistedTailCall.

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

The x86 JIT, on the other hand, currently only attempts to do a tail call if the IL explicitly uses the "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 prefix, but the C++ and F# compilers generate it automatically, so the programmer has very little control over this choice. So unless you write in IL, or use some form of IL rewriter, your ability to add or remove the ".tail" prefix is limited at best.

Lastly if you're still reading you have probably noticed that there is redundancy. This is partly because the messages are generated by different components in the runtime - the VM, the x86 JIT, and the x64 JIT, which were developed, and have evolved, fairly independently. There is also some amount of redundancy as a safety precaution.

Grant RIchins  
CLR Codegen Team

## Testing, Testing: Hey, is this thing on?

September 24, 2013, 1:15 pm

[» Next: RyuJIT CTP1 is available for public consumption!](#)

[« Previous: JIT ETW Tail Call Event Fail Reasons](#)



Hey folks! For the persevering Codegen aficionados (I was going to say junkies, but aficionado seems much more accurate to me) out there, you may have noticed that there hasn't been anything posted here in some time. Well, that'll be changing soon. I've been working on a (completely self-serving) write up about floating point, but that's really mostly a tangent, useful for pointing unsuspecting bug reporters at when I resolve their bug reports as "by design". The real point is that there are Big Things coming. And you can get all sorts of details here. Real soon. Seriously.

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

Viewing all 40 articles Page 1 [»](#) [Remove ADS](#)

[Browse latest](#) [View live](#)

[More Pages to Explore ....+](#)

RSSING>> LATEST POPULAR TOP RATED TRENDING

© 2024 //www.rssing.com