عنوان: **توصیه هایی در استفاده از NGEN**
نویسنده: م منفرد
تاریخ: ۱۳۹۲/۱۱/۲۷ ۹:۰
آدرس: www.dotnettips.info
گروه‌ها: Performance, JIT, NGEN

یکی از مسائل ریز و فنی در دنیای .NET استفاده یا عدم استفاده از NGEN است. در مقاله کوتاه زیر بهترین مکان‌های استفاده و عدم استفاده از آن را در چند بند خلاصه می‌کنم:

کجا از NGEN استفاده کنیم؟

برنامه هایی که مقدار زیادی کد مدیریت شده قبل از نمایش فرم برنامه دارند. مانند برنامه Blend که مقدار زیادی کد در شروع برنامه دارد. استفاده از ngen می‌تواند باعث افزایش کارایی و سرعت اجرای برنامه شود

فریم ورک ها، dllها و کامپوننت‌های عمومی: کدهای تولید شده توسط JIT قابل اشتراک بین برنامه‌های مختلف نیستند ولی NGEN قابل اشتراک مابین برنامه‌های مختلف می‌باشد. بنابراین اگر کامپوننتی دارید که در بین برنامه‌های مختلف مشترک استفاده می‌شود، این کار می‌تواند سرعت شروع برنامه‌ها را بالا برده استفاده از منابع سیستم را کاهش دهد

برنامه هایی که در terminal serverها استفاده می‌شوند: توضیح فوق در مورد این برنامه‌ها نیز صادق است.

کجا از NGEN استفاده نکنیم؟

برنامه‌های کوچک: عملا سرعت JIT آن قدر بالا است که NGEN کار را کندتر خواهد کرد!

برنامه‌های سروری که سرعت شروع آن مهم نیست: برنامه‌ها یا dll هایی که سرعت شروع آنها مهم نیست، اگر NGEN نشوند سرعت بیشتری برای شما به ارمغان خواهند داشت چون JIT در هنگام اجرا، کد را بهینه می‌کند ولی NGEN این کار را انجام نمی‌دهد.

چند نکته دیگر که باید در نظر داشته باشید این است که قرار نیست NGEN مثل یک جادوگر کد شما را جادو کند که سریع‌تر اجرا شود. تنها کد را از قبل به کد native مربوط به معماری cpu شما کامپایل خواهد کرد که شروع اجرای آن سریع‌تر شود. البته این جادوگر :) قربانی هم می‌خواهد. قربانی آن optimizationهای داخلی JIT است که در برنامه شما اعمال نخواهد شد. بنابراین در رابطه با استفاده از NGEN نهایت دقت را به خرج دهید.

نظرات خوانندگان

نویسنده: شهروز جعفری
تاریخ: ۱۳۹۲/۱۱/۲۷ ۱۶:۲۲

بد نیست به این مطلب (NGEN.EXE در دات نت) یک سری بزنید.

نویسنده: ناصرنیازی
تاریخ: ۱۳۹۲/۱۱/۲۹ ۱۹:۲۸

می خواستم بپرسم آیا می‌توان با این برنامه فایل اجرایی برنامه را Standalone کرد مثل فایل‌های دلفی۷ که همه جا اجرا میشه ؟

نویسنده: م منفرد
تاریخ: ۱۳۹۲/۱۱/۳۰ ۰:۳۰

خیر. برای اجرای برنامه هنوز نیاز به فریمورک دات نت دارید.

عنوان: **فعال سازی Multicore JIT**
نویسنده: سیروان عفیفی
تاریخ: ۱۳۹۳/۰۴/۲۸ ۱۷:۵۵
آدرس: www.dotnettips.info
گروه‌ها: Performance, JIT, .NET 4.5, MultiCore JIT

Multicore JIT یکی از قابلیت‌های کلیدی در دات نت 4.5 می‌باشد که در واقع راه حلی برای بهبود سرعت اجرای برنامه‌های دات نتی است. قبل از معرفی این قابلیت ابتدا اجازه دهید نحوه کامپایل یک برنامه دات نتی را بررسی کنیم.

انواع compilation

در حالت کلی دو نوع فرآیند کامپایل داریم:

Explicit

در این حالت دستورات قبل از اجرای برنامه به زبان ماشین تبدیل می‌شوند. به این نوع کامپایلرها AOT یا Ahead Of Time گفته می‌شود. این نوع از کامپایلرها برای اطمینان از اینکه CPU بتواند قبل از انجام تعاملی تمام خطوط کد را تشخیص دهد، طراحی شده اند.

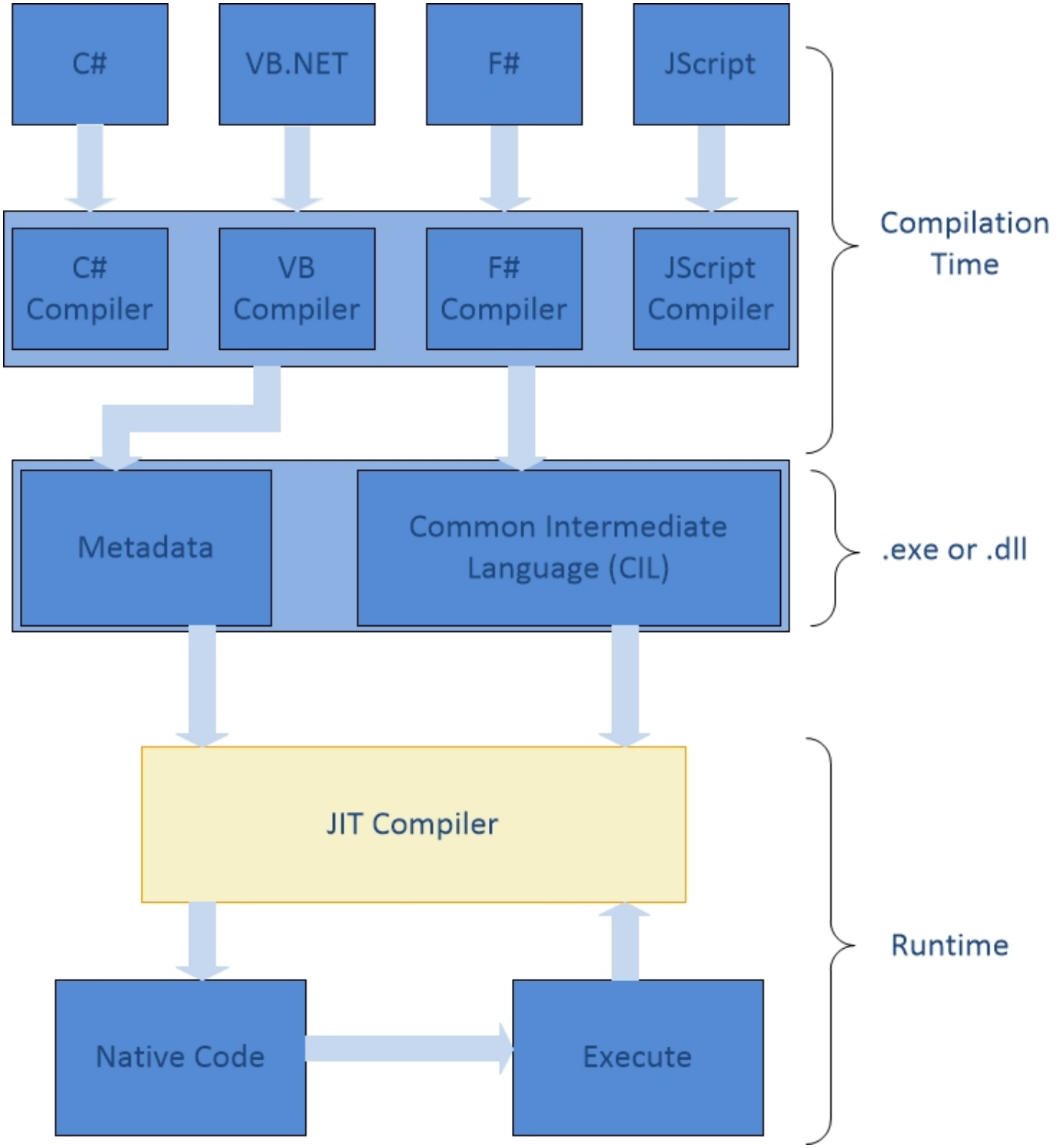
Implicit

این نوع compilation به صورت دو مرحله ایی صورت می‌گیرد. در اولین قدم سورس کد توسط یک کامپایلر به یک زبان سطح میانی(IL) تبدیل می‌شود. در مرحله بعدی کد IL به دستورات زبان ماشین تبدیل می‌شوند. در دات نت فریم ورک به این کامپایلر JIT یا Just-In-Time گفته می‌شود.

در حالت دوم قابلیت جابجایی برنامه به آسانی امکان پذیر است، زیرا اولین قدم از فرآیند به اصطلاح platform agnostic می‌باشد، یعنی قابلیت اجرا بر روی گستره وسیعی از پلت فرم‌ها را دارد.

کامپایلر JIT

JIT بخشی از Common Language Runtime یا CLR می‌باشد. CLR در واقع وظیفه مدیریت اجرای تمام برنامه‌های دات نتی را برعهده دارد.

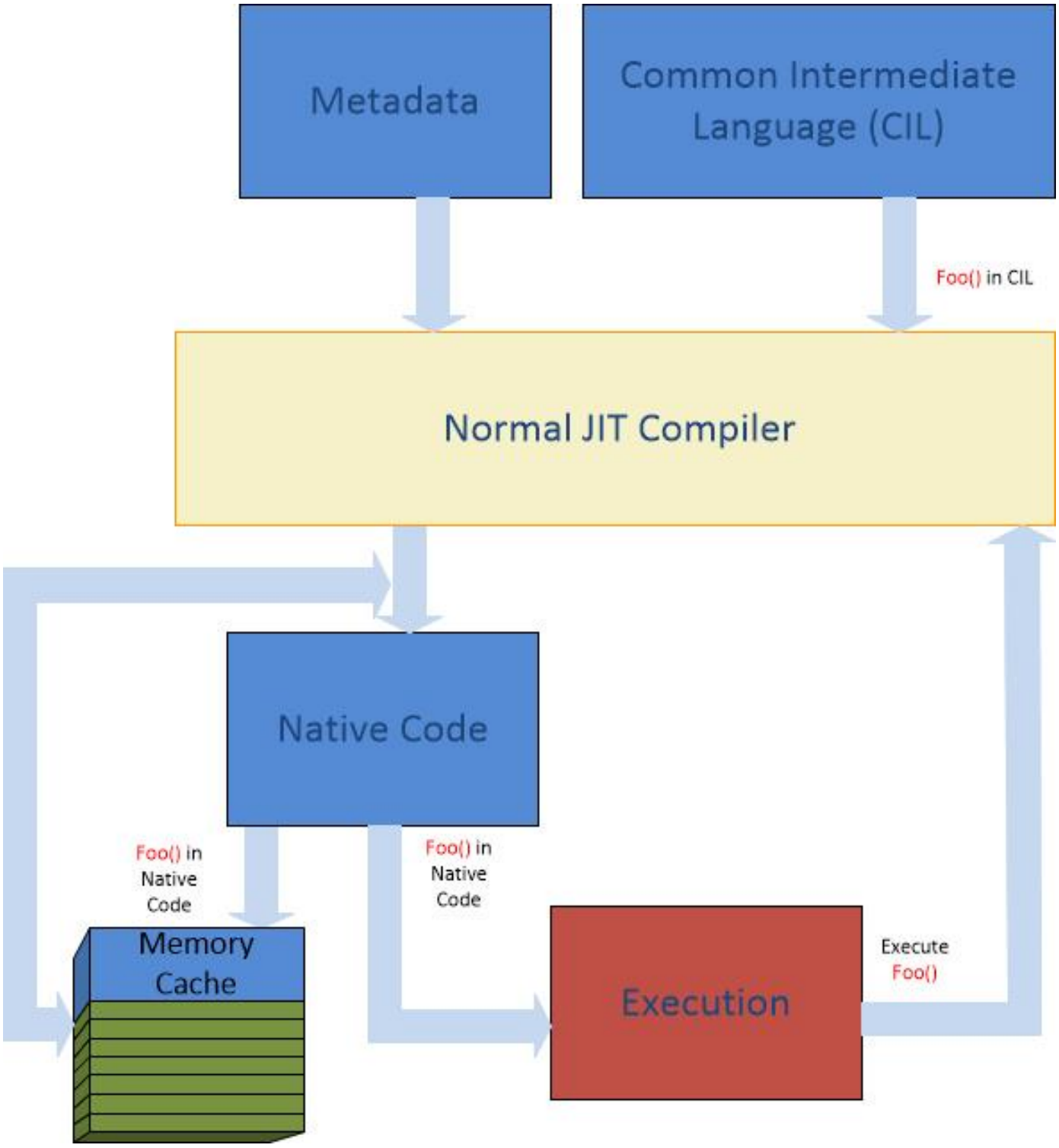


همانطور که در تصویر فوق مشاهده می‌کنید، سورس کد توسط کامپایلر دات نت به exe و یا dll کامپایل می‌شود. کامپایلر JIT تنها متدهایی را که در زمان اجرا(runtime) فراخوانی می‌شوند را کامپایل می‌کند. در دات نت فریم ورک سه نوع JIT Compilation داریم:

Normal JIT Compilation

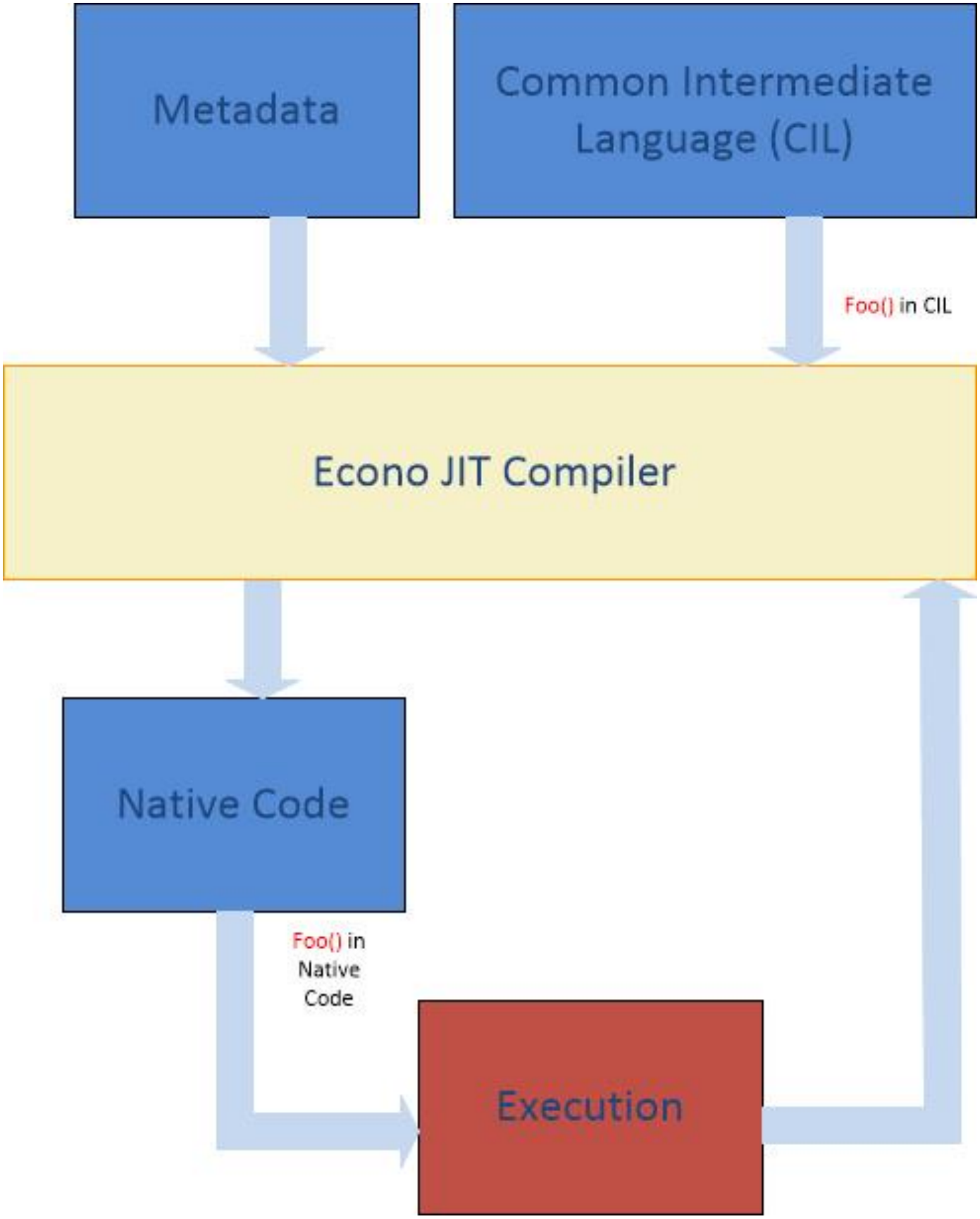
در این نوع کامپایل، متدها در زمان فراخوانی در زمان اجرا کامپایل می‌شوند. بعد از اجرا، متد داخل حافظه ذخیره می‌شود. به متدهای ذخیره شده در حافظه jitted گفته می‌شود. دیگر نیازی به کامپایل متد jit شده نیست. در فراخوانی بعدی، متد مستقیماً

از حافظه کش در دسترس خواهد بود.



Econo JIT Compilation

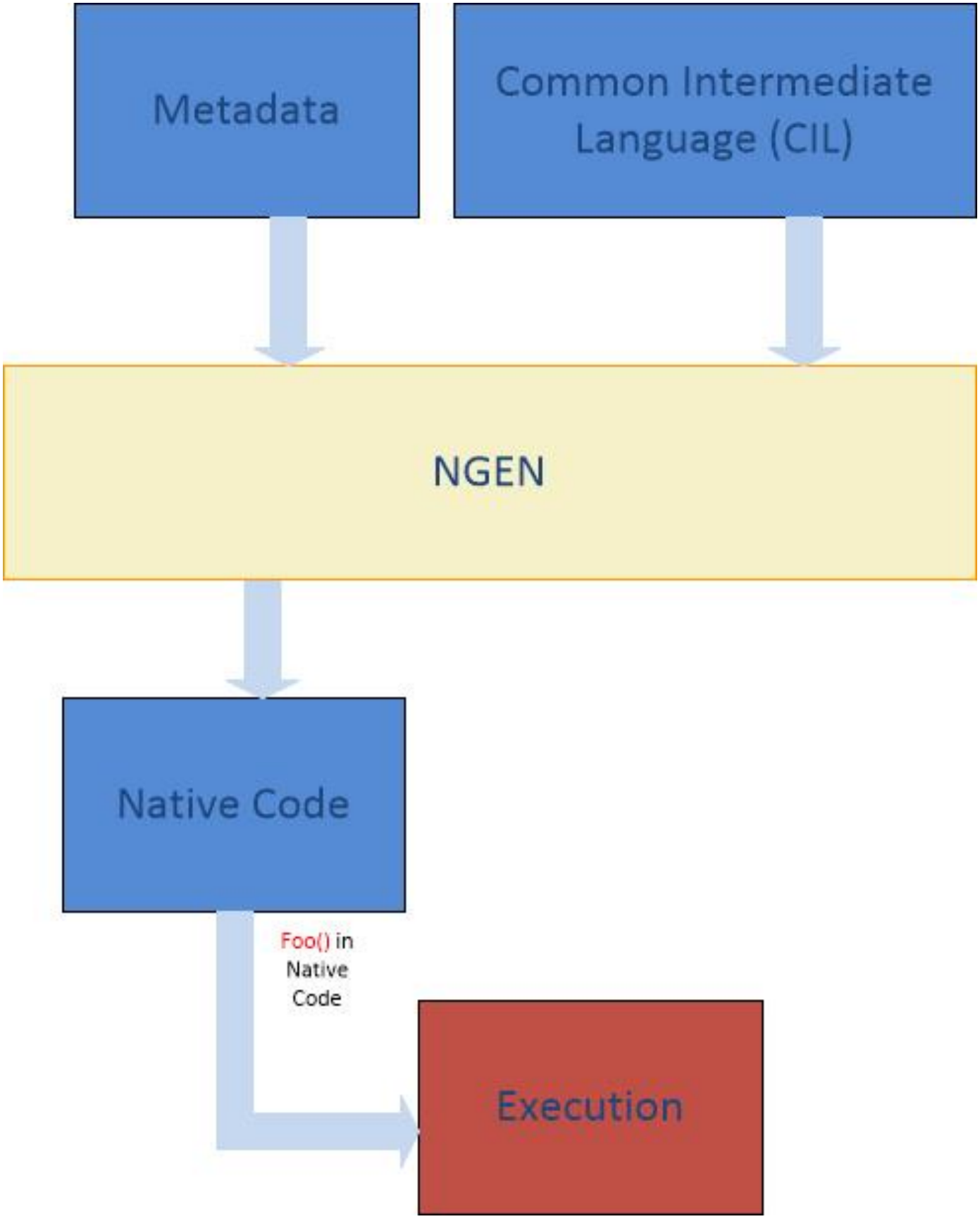
این نوع کامپایل شبیه به حالت Normal JIT است با این تفاوت که متدها بلافاصله بعد از اجرا از حافظه حذف می‌شوند.



Pre-JIT Compilation

یکی دیگر از حالت‌های کامپایل برنامه‌های دات نتی Pre-JIT Compilation می باشد. در این حالت به جای متدهای مورد استفاده،

کل اسمبلی کامپایل می‌شود. در دات نت می‌توان اینکار را توسط Ngen.exe یا (Native Image Generator) انجام داد. تمام دستورالعمل‌های CIL قبل از اجرا به کد محلی(Native Code) کامپایل می‌شوند. در این حالت runtime می‌تواند از native images به جای کامپایلر JIT استفاده کند. این نوع کامپایل عملیات تولید کد را در زمان اجرای برنامه به زمان Installation منتقل می‌کند، در اینصورت برنامه نیاز به یک Installer برای اینکار دارد.

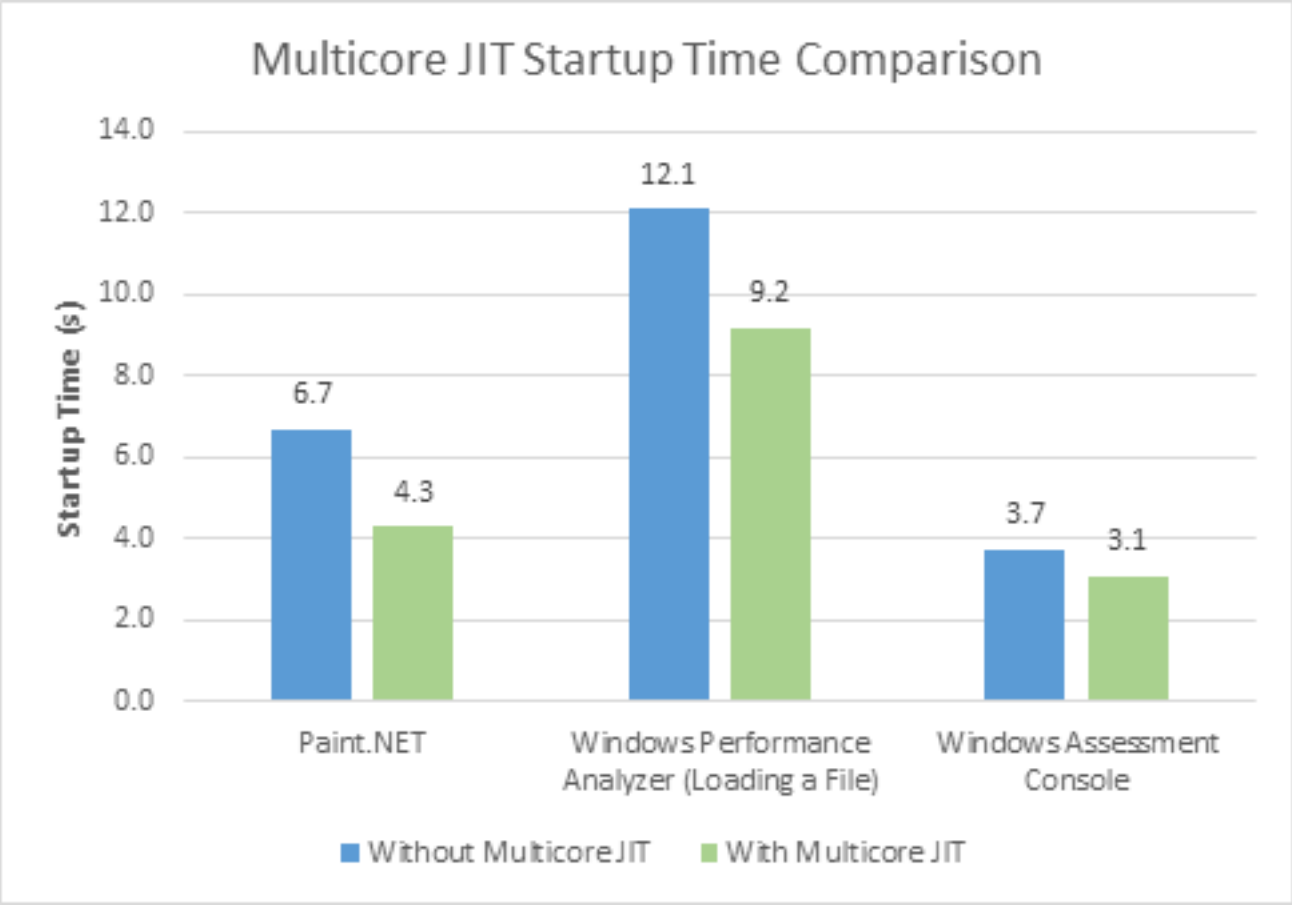


Multicore JIT

در دات نت فریم ورک 4.5 یک راه حل جایگزین دیگر برای بهینه سازی و بهبود سرعت اجرای برنامه‌های دات نت وجود دارد.

همانطور که عنوان شد Ngen.exe برای در دسترس بودن نیاز به Installer برای برنامه دارد. توسط Multicore JIT متدها بر روی دو هسته به صورت موازی کامپایل می‌شوند، در اینصورت می‌توانید تا 50 درصد از JIT Time صرفه جویی کنید.

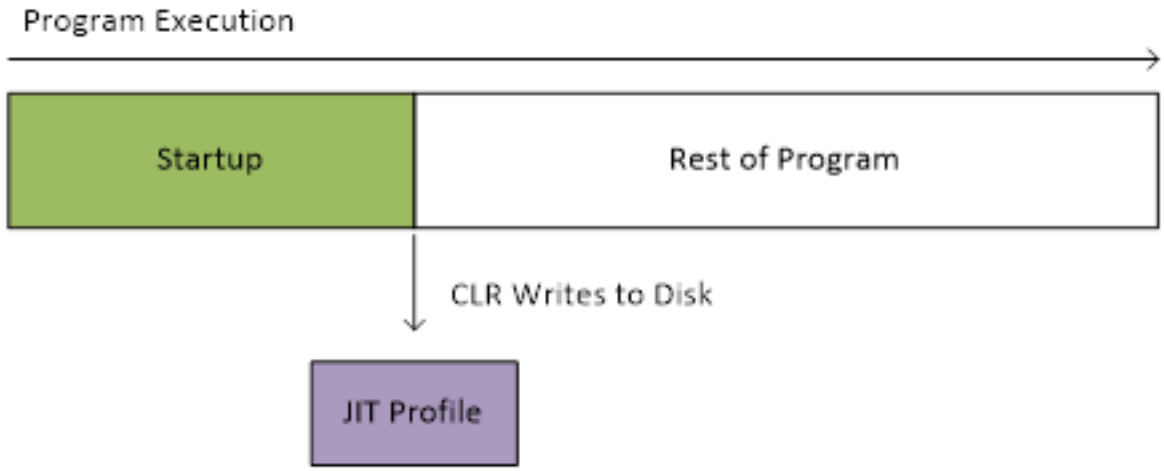
Multicore JIT همچنین می‌تواند باعث بهبود سرعت در برنامه‌های WPF شود. در نمودار زیر می‌توانید حالت‌های استفاده و عدم استفاده از Multicore JIT را در سه برنامه WPF نوشته شده مشاهده کنید.



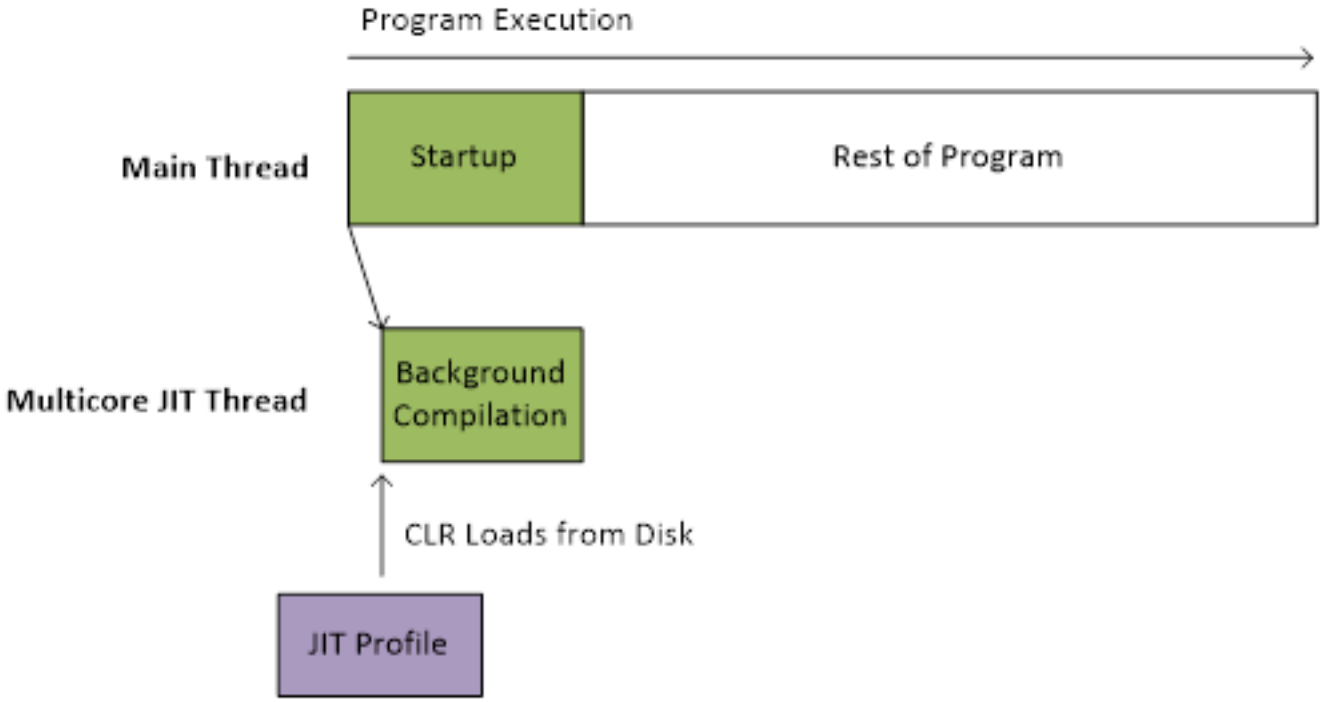
Multicore JIT در عمل

Multicore JIT از دو مد عملیاتی استفاده می‌کند: مد ثبت(Recording mode)، مد بازپخش(Playback mode)

در حالت ثبت کامپایلر JIT هر متدی که نیاز به کامپایل داشته باشد را رکورد می‌کند. بعد از اینکه CLR تعیین کند که اجرای برنامه به اتمام رسیده است، تمام متدهایی که اجرا شده اند را به صورت یک پروفایل بر روی دیسک ذخیره می‌کند.



هنگامیکه Multicore JIT فعال می‌شود، با اولین اجرای برنامه، حالت ثبت مورد استفاده قرار می‌گیرد. در اجراهای بعدی، از حالت بازپخش استفاده می‌شود. حالت بازپخش پروفایل را از طریق دیسک بارگیری کرده، و قبل از اینکه این اطلاعات توسط ترد اصلی مورد استفاده قرار گیرد، از آنها برای تفسیر (کامپایل) متدها در پیش‌زمینه استفاده می‌کند.



در نتیجه، ترد اصلی به کامپایل دیگری نیاز ندارد، در این حالت سرعت اجرای برنامه بیشتر می‌شود. حالت‌های ثبت و بازپخش تنها برای کامپیوترهایی با چندین هسته فعال می‌باشند.

استفاده از Multicore JIT

در برنامه‌های ASP.NET 4.5 و Silverlight 5 به صورت پیش فرض این ویژگی فعال می‌باشد. ازآنجائیکه این برنامه‌ها hosted application هستند؛ در نتیجه فضای مناسبی برای ذخیره سازی پروفایل در این نوع برنامه‌ها موجود می‌باشد. اما برای برنامه‌های

Desktop این ویژگی باید فعال شود. برای اینکار کافی است دو خط زیر را به نقطه شروع برنامه تان اضافه کنید:

```
public App()
{
    ProfileOptimization.SetProfileRoot(@"C:\MyAppFolder");
    ProfileOptimization.StartProfile("Startup.Profile");
}
```

توسط متد SetProfileRoot می‌توانیم مسیر ذخیره سازی پروفایل JIT را مشخص کنیم. در خط بعدی نیز توسط متد StartProfile نام پروفایل را برای فعال سازی Multicore JIT تعیین می‌کنیم. در این حالت در اولین اجرای برنامه پروفایلی وجود ندارد، Multicore JIT در حالت ثبت عمل می‌کند و پروفایل را در مسیر تعیین شده ایجاد می‌کند. در دومین بار اجرای برنامه CRL پروفایل را از اجرای قبلی برنامه بارگذاری می‌کند؛ در این حالت Multicore JIT به صورت بازپخش عمل می‌کند.

همانطور که عنوان شد در برنامه‌های ASP.NET 4.5 و Silverlight 5 قابلیت Multicore JIT به صورت پیش فرض فعال می‌باشد. برای غیر فعال سازی آن می‌توانید با تغییر فلگ profileGuidedOptimizations به None اینکار را انجام دهید:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <!-- ... -->
 <system.web>
 <compilation profileGuidedOptimizations="None" />
 <!-- ... -->
 </system.web>
</configuration>
```

عنوان: .NET Just-In-Time Optimization
نویسنده: غلامرضا ربال
تاریخ: ۱۳۹۴/۰۱/۰۵ ۱۰:۳۰
آدرس: www.dotnettips.info
گروه‌ها: Performance, JIT, .NET 4.5

هدف از توابع خطی(Inline)

استفاده از توابع، مقداری بر زمان اجرای برنامه می‌افزاید؛ هرچند که این زمان بسیار کم و در حد میلی ثانیه است، اما باری را بر روی برنامه قرار می‌دهد و علت این تاخیر زمانی این است که در فراخوانی و اعلان توابع، کامپایلر یک کپی از تابع مورد نظر را در حافظه قرار می‌دهد و در فراخوانی تابع، به آدرس مذکور مراجعه می‌کند و در عین حال آدرس موقعیت توقف دستورات در تابع main را نیز ذخیره می‌کند تا پس از پایان تابع، به آدرس قبل برگردد و ادامه‌ی دستورات را اجرا کند. در نتیجه این آدرس دهی‌ها و نقل و انتقالات بین آنها بار زمانی را در برنامه ایجاد می‌کند که در صورت زیاد بودن توابع در برنامه و تعداد فراخوانی‌های لازم، زمان قابل توجهی خواهد شد.

یکی از تکنیک‌های بهینه که برای کاهش زمان اجرای برنامه توسط کامپایلرها استفاده می‌شود استفاده از توابع خطی (inline) است. این امکان در زبان C با عنوان توابع ماکرو(Macro function) و در C++ با عنوان توابع خطی (inline function) وجود دارد.

در واقع توابع خطی به کامپایلر پیشنهاد می‌دهند، زمانی که سربار فراخوانی تابع بیشتر از سربار بدنه خود متد باشد، برای کاهش هزینه و زمان اجرای برنامه از تابع به صورت خطی استفاده کند و یک کپی از بنده‌ی تابع را در قسمتی که تابع ما فراخوانی شده است، قرار دهد که مورد آدرس دهی از میان خواهد رفت!

نمونه ای از پیاده سازی این تکنیک در زبان C++ :

```
inline type name(parameters)
{
    ...
}
```

بررسی متدهای خطی در سی شارپ به مثال زیر توجه کنید:

قسمت‌های getter و setter مربوط به پراپرتی‌ها سربار اضافی بر کلاس Vector می‌افزایند. این موضوع شاید آنچنان مسئله‌ی مهمی نباشد. ولی فرض کنید این پراپرتی‌ها به شکل زیر داخل حلقه‌ای طولانی قرار گیرند. اگر با استفاده از یک پروفایلر زمان اجرای برنامه را زیر نظر بگیرید، خواهید دید که بیش از 90 درصد آن صرف فراخوانی‌های متدهای بخش‌های get , set پراپرتی‌ها است. برای این منظور باید مطمئن شویم که فراخوانی این متدها، به صورت خطی صورت می‌گیرد!

```
public class Vector
  {
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }

    // ...
  }
```

برای این منظور آزمایشی را انجام می‌دهیم. فرض کنید کلاسی را به شکل زیر داشته باشیم:

```
public class MyClass
  {
    public int A { get; set; }
    public int C;
  }
```

و برای استفاده از آن به شکل زیر عمل کنیم:

```
static void Main()
{
    MyClass target = new MyClass();
    int a = target.A;
    Console.WriteLine("A = {0}", a);
    int c = target.C;
```

```
    Console.WriteLine("C = {0}", c);
  }
```

بعد از دیباگ برنامه و مشاهده‌ی کدهای ماشین مربوط به آن خواهیم دید که متد مربوط به getter پراپرتی A به صورت خطی فراخوانی نشده است:

```
int a = target.A;
  0000003e  mov         ecx,edi
  00000040  cmp         dword ptr [ecx],ecx
  00000042  call        dword ptr ds:[05FA29A8h]
  00000048  mov         esi,eax
  0000004a  mov         dword ptr [esp+4],esi
             int c = target.C;
  00000098  mov         edi,dword ptr [edi+4]
    MyClass.get_A() looks like this:
  00000000  push        esi
  00000001  mov         esi,ecx
  00000003  cmp         dword ptr ds:[03B701DCh],0
  0000000a  je          00000011
  0000000c  call        76BA6BA7
  00000011  mov         eax,dword ptr [esi+0Ch]
  00000014  pop         esi
  00000015  ret
```

چه اتفاقی افتاده است؟

کامپایلر سی شارپ در زمان کامپایل، کدهای برنامه را به کدهای IL تبدیل می‌کند و JITکامپایلر، این کدهای IL را گرفته و کد ساده‌ی ماشین را تولید می‌کند. لذا به دلیل اینکه JIT با معماری پردازنده آشنایی کافی دارد، مسئولیت تصمیم گیری اینکه کدام متد به صورت خطی فراخوانی شود برعهده‌ی آن است. در واقع این JIT است که تشخیص می‌دهد که آیا فراخونی متد به صورت خطی مناسب است یا نه و به صورت یک معاوضه کار بین خط لوله دستورالعمل‌ها و کش است.

اگر شما برنامه‌ی خود را با (F5) و همگام با دیباگ اجرا کنید، تمام بهینه سازی‌های JIT که Inline Method هم یکی از آنهاست، از کار خواهند افتاد. برای مشاهده‌ی کد بهینه شده باید با بدون دیباگ (CTRL+F5) برنامه خود را اجرا کنید که در آن صورت مشاهده خواهید کرد، متد getter مربوط به پراپرتی A به صورت خطی استفاده شده است.

```
int a = target.A;
00000024  mov         ebx,dword ptr [edi+0Ch]
```

JIT محدودیت هایی برای فراخونی به صورت خطی متدها دارد :

متد هایی که حجم کد IL آنها بیشتر از 32 بایت است.

متدهای بازگشتی.

متدهایی که با اتریبیوت MethodImpl علامتگذاری شدند و MethodImplOptions.NoInlining اعمال شده بر آن

متدهای virtual

متدهایی که دارای کد مدیریت خطا هستند

Methods that take a large value type as a parameter

Methods with complicated flowgraphs

برای اینکه در سی شارپ به کامپایلر اعلام کنیم تا متد مورد نظر به صورت خطی مورد استفاده قرار گیرد، در دات نت 4.5 توسط اتریبیوت MethodImpl و اعمال MethodImplOptions.AggressiveInlining که یک نوع شمارشی است می‌توان این کار را انجام داد. مثال:

```
using System;
using System.Diagnostics;
using System.Runtime.CompilerServices;
class Program
{
    const int _max = 10000000;
    static void Main()
```

```
    {
         // ... Compile the methods.
        Method1();
        Method2();
        int sum = 0;
        var s1 = Stopwatch.StartNew();
        for (int i = 0; i < _max; i++)
           {
              sum += Method1();
           }
         s1.Stop();
         var s2 = Stopwatch.StartNew();
         for (int i = 0; i < _max; i++)
             {
                sum += Method2();
             }
         s2.Stop();
         Console.WriteLine(((double)(s1.Elapsed.TotalMilliseconds * 1000000) /
        _max).ToString("0.00 ns"));
        Console.WriteLine(((double)(s2.Elapsed.TotalMilliseconds * 1000000) /
      _max).ToString("0.00 ns"));
        Console.Read();
      }
      static int Method1()
      {
         // ... No inlining suggestion.
         return "one".Length + "two".Length + "three".Length +
              "four".Length + "five".Length + "six".Length +
              "seven".Length + "eight".Length + "nine".Length +
               "ten".Length;
    }
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    static int Method2()
    {
        // ... Aggressive inlining.
        return "one".Length + "two".Length + "three".Length +
           "four".Length + "five".Length + "six".Length +
           "seven".Length + "eight".Length + "nine".Length +
            "ten".Length;
    }
}
Output
7.34 ns    No options
0.32 ns    MethodImplOptions.AggressiveInlining
```

در واقع با اعمال این اتریبیوت، محدودیت شماره یک مبنی بر محدودیت حجم کد IL مربوط به متد، در نظر گرفته نخواهد شد .

مطالعه بیشتر: http://dotnet.dzone.com/news/aggressive-inlining-clr-45-jit

/http://www.ademiller.com/blogs/tech/2008/08/c-inline-methods-and-optimization

http://www.dotnetperls.com/aggressiveinlining

http://blogs.msdn.com/b/ericgu/archive/2004/01/29/64644.aspx

https://msdn.microsoft.com/en-us/library/ms973858.aspx#highperfmanagedapps_topic10