Viewing JIT Dumps

This document is intended for people interested in seeing the disassembly, GC info, or other details the JIT generates for a managed program.

To make sense of the results, it is recommended you also read the Reading a JitDump section of the RyuJIT Overview.

Setting up our environment

The first thing to do is setup the .NET Core app we want to dump. Here are the steps to do this, if you don't have one ready:

- Cd into `src/coreclr`
- Perform a release build of the runtime by passing `release` to the build command. You don't need to build tests, so you can pass `skiptests` to the build command to make it faster. Note: the release build can be skipped, but in order to see optimized code of the core library it is needed.
- Perform a debug build of the runtime. Tests aren't needed as in the release build, so you can pass `skiptests` to the build command. Note: the debug build is necessary, so that the JIT recognizes the configuration knobs.
- Install the (latest) .NET CLI, which we'll use to compile/publish our app.
- `cd` to where you want your app to be placed, and run `dotnet new console`.
- Modify your `csproj` file so that it contains a RID (runtime ID) corresponding to the OS you're using in the `<RuntimeIdentifier>` tag. For example, for Windows 10 x64 machine, the project file is:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <RuntimeIdentifier>win-x64</RuntimeIdentifier>
  </PropertyGroup>

</Project>
```

  You can find a list of RIDs and their corresponding OSes here.

- Edit `Program.cs`, and call the method(s) you want to dump in there. Make sure they are, directly or indirectly, called from `Main`. In this example, we'll be looking at the disassembly of our custom function `InefficientJoin`:

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine(InefficientJoin(args));
        }

        // Add NoInlining to prevent this from getting
        // mixed up with the rest of the code in Main
        [MethodImpl(MethodImplOptions.NoInlining)]
        private static string InefficientJoin(IEnumerable<string> ar
        {
            var result = string.Empty;
            foreach (var arg in args) result += (arg + ' ');
            return result.Substring(0, Math.Max(0, result.Length - 1
        }
    }
}
```

- After you've finished editing the code, run `dotnet restore` and `dotnet publish -c Release`. This should drop all of the binaries needed to run your app in `bin/Release/net5.0/<rid>/publish`.

- Overwrite the CLR dlls with the ones you've built locally. If you're a fan of the command line, here are some shell commands for doing this:

```
# Windows
robocopy /e <runtime-repo path>\artifacts\bin\coreclr\Windows_NT.<ar
copy /y <runtime-repo path>\artifacts\bin\coreclr\Windows_NT.<arch>.

# Unix
cp -rT <runtime-repo path>/artifacts/bin/coreclr/<OS>.<arch>.Release
cp <runtime-repo path>/artifacts/bin/coreclr/<OS>.<arch>.Debug/libcl
```

- Set the configuration knobs you need (see below) and run your published app. The info you want should be dumped to stdout.

  Here's some sample output on my machine showing the disassembly for `InefficientJoin`:

```
G_M2530_IG01:
       55                   push     rbp
       4883EC40             sub      rsp, 64
```



```
G_M2530_IG02:
       49BB60306927E5010000 mov      r11, 0x1E527693060
       4D8B1B               mov      r11, gword ptr [r11]
       4C895DF8             mov      gword ptr [rbp-08H], r11
       49BB200058F7FD7F0000 mov      r11, 0x7FFDF7580020
       3909                 cmp      dword ptr [rcx], ecx
       41FF13               call     gword ptr [r11]System.Collectio
       488945F0             mov      gword ptr [rbp-10H], rax

; ...
```

Setting configuration variables

GuardedDevirtualization.md
JitOptimizerPlanningGuide....
JitOptimizerTodoAssessmen...
Perf-Score.md
arm64-jit-frame-layout.md
eh-writethru.md
finally-optimizations.md
first-class-structs.md
inline-size-estimates.md
inlining-plans.md
jit-call-morphing.md
longs-on-32bit-arch.md
lsra-detail.md
lsra-throughput.md
multi-reg-call-nodes.md
object-stack-allocation.md
porting-ryujit.md
removing-embedded-state...
ryujit-overview.md
ryujit-tutorial.md
struct-abi.md
variabletracking.md
viewing-jit-dumps.md
profiling
features
specs
issue-mappings
manpages
project
workflow
README.md
area-owners.md
deep-dive-blog-posts.md

The behavior of the JIT can be controlled via a number of configuration variables. These are declared in inc/clrconfigvalues.h and jit/jitconfigvalues.h. When used as an environment variable, the string name generally has `COMPlus_` prepended. When used as a registry value name, the configuration name is used directly.

These can be set in one of three ways:

- Setting the environment variable `COMPlus_<flagname>`. For example, the following will set the `JitDump` flag so that the compilation of all methods named `Main` will be dumped:

```
# Windows
set COMPlus_JitDump=Main

# Powershell
$env:COMPlus_JitDump="Main"

# Unix
export COMPlus_JitDump=Main
```

- *Windows-only:* Setting the registry key `HKCU\Software\Microsoft\.NETFramework`, Value `<flagName>`, type `REG_SZ` or `REG_DWORD` (depending on the flag).
- *Windows-only:* Setting the registry key `HKLM\Software\Microsoft\.NETFramework`, Value `<flagName>`, type `REG_SZ` or `REG_DWORD` (depending on the flag).

Specifying method names

The complete syntax for specifying a single method name (for a flag that takes a method name, such as `COMPlus_JitDump`) is:

```
[[<Namespace>.]<ClassName>::]<MethodName>[([<types>)]
```

For example

```
System.Object::ToString(System.Object)
```

The namespace, class name, and argument types are optional, and if they are not present, default to a wildcard. Thus stating:

```
Main
```

will match all methods named Main from any class and any number of arguments.

`<types>` is a comma separated list of type names. Note that presently only the number of arguments and not the types themselves are used to distinguish methods. Thus, `Main(Foo, Bar)` and `Main(int, int)` will both match any main method with two arguments.

The wildcard character `*` can be used for `<ClassName>` and `<MethodName>`. In particular `*` by itself indicates every method.

Useful COMPlus variables

Below are some of the most useful `COMPlus` variables. Where {method-list} is specified in the list below, you can supply a space-separated list of either fully-qualified or simple method names (the former is useful when running something that has many methods of the same name), or you can specify `*` to mean all methods.

- `COMPlus_JitDump`={method-list} – dump lots of useful information about what the JIT is doing. See Reading a JitDump for more on how to analyze this data.

- `COMPlus_JitDumpASCII` ={1 or 0} - Specifies whether the JIT dump should be ASCII only (Defaults to 1). Disabling this generates more readable expression trees.
- `COMPlus_JitDisasm` ={method-list} – dump a disassembly listing of each method.
- `COMPlus_JitDiffableDasm` – set to 1 to tell the JIT to avoid printing things like pointer values that can change from one invocation to the next, so that the disassembly can be more easily compared.
- `COMPlus_JitGCDump` ={method-list} – dump the GC information.
- `COMPlus_JitUnwindDump` ={method-list} – dump the unwind tables.
- `COMPlus_JitEHDump` ={method-list} – dump the exception handling tables.
- `COMPlus_JitTimeLogFile` ={file name} – this specifies a log file to which timing information is written.
- `COMPlus_JitTimeLogCsv` ={file name} – this specifies a log file to which summary timing information can be written, in CSV form.

Dumping native images

If you followed the tutorial above and ran the sample app, you may be wondering why the disassembly for methods like `Substring` didn't show up in the output. This is because `Substring` lives in mscorlib, which (by default) is compiled ahead-of-time to a native image via crossgen. Telling crossgen to dump the info works slightly differently.

- First, perform a debug build of the native parts of the repo: `build skipmscorlib skiptests`.
  - This should produce the binaries for crossgen in `artifacts/Product/<OS>.<arch>.Debug`.
- Next, set the appropriate configuration knob for the info you want to dump. Usually, this is just the same as the corresponding JIT knob, except prefixed with `Ngen`; for example, to show the disassembly listing of a particular method you would `set COMPlus_NgenDisasm=Foo`.
- Run crossgen on the assembly you want to dump: `crossgen MyLibrary.dll`
  - If you want to see the output of crossgen specifically for mscorlib, invoke `build skipnative skiptests` from the repo root. The dumps should be written to a file in `artifacts/Logs` that you can just view.