PERFORMANCE VERSES CODE

AN ANALYSIS OF THE CODE PERFORMANCE OF DIFFERENT LANGUAGES/FRAMEWORKS ON RUNTIME SYSTEMS

Part 2 of a two part series on Application Performance under Windows 10.

Tim Mangan
TMurgent Technologies, LLP
March, 2016

ABSTRACT

This paper is from a two part series looking at performance aspects of application software on Windows 10. Part 1 looks at data while Part 2 looks at code.

This paper, Part 2, attempts to quantify the some of the performance impacts on a Microsoft operating system caused by the selection of application development language and frameworks. Specifically, we focusing on the generated code, this paper looks at the performance impacts of:

- Footprint
- Memory Use
- Memory Sharing/Reuse
- CPU
- Time of Execution

This analysis is done using custom built software in a variety of forms for comparison:

- "Unmanaged Win32" C++ code
- .Net C# code in IL form
- .Net C# code in native compiled form
- Windows Universal App C# code

The results of this testing did not exactly match our expectations prior to testing.

INTRODUCTION

How does the choice of programming language and frameworks affect the performance of the application on a runtime system, and impact on other software running on the same system?

This question came to mind while researching some of the changes made by Microsoft in Windows 10 updates in the Fall of 2015 (which led to the development of Part 1 of this series). I had detected some differences in operating system behavior by different apps, and upon investigation determined that the choice of programing languages and development frameworks were the cause of the differences that I had detected.

Through my experiences of developing applications and testing performance of systems, I had rough ideas on how I expected the choices to impact performance, but I had never tested it. Certainly, every time Microsoft came up with something new they always claimed how it was better than what came before, but it usually involved how it impacted the developer and not the end-user experience. My expectations were that all of the nice new features to take care of stuff for the developer would have to come at a cost, especially of CPU that we have so much of that it is OK to waste it (a little) now. So I assumed any such test would show an overhead when you used the new-and-improved languages and frameworks. Testing has caused me to change my mind a little.

TESTING METHODOLOGY

When I test performance and behavior of the OS, I like to do it in a big way and with precision. So off the shelf software is not used to induce behavior. I write an app specific to the purpose where I know exactly what every line does. I write a really, really, bad app. One that does what a normal app might do a little of, just in a big way. Often, I can use existing monitoring software, written by myself of others, but maybe I'll have to enhance the monitoring software to capture exactly what I want to show.

In this case, I started with a bad application tool and monitor that I had written to investigate performance impacts a dozen years ago, along with a monitoring tool written to investigate Vista memory management and modified these to meet this challenge.

Tests are always run on semi-isolated and rebooted systems given time to settle down the typical post boot and post logon activities of the OS. The tests are also repeated enough times to ensure that nothing unexpected crept into the test.

TEST DESCRIPTIONS

The tests in this paper were created using as-similar-as-possible code from application projects built using three era of languages/platforms:

- A "Win32" style application written in C++. The app will actually be a 64-bit app for consistency with other tests. The C++ code is "unmanaged", meaning that it is up to the application to manage its own memory.
- A "Microsoft.Net" style application written in C#. The app will use a WPF front end to present a user interface, but the dll under test is just some plain C# code. The Microsoft.Net 4.5.2 Framework was used. The dll and exe were built as "AnyCPU" without the 32bit-preference set, so that they would execute as x64 on the test platform.
- A "Windows Universal" application written in C# (actually an identical copy of the C# code of the previous example). The app uses a Universal App front end and the Windows RT framework. The dll and exe were built as x64.

The second case was later broken into two separate cases, one in which NGEN compilation of the code into native code was performed and one where it was not. NGEN compilation is something typically requested as part of software installation, however sometimes developers forget to include this in the setup project and sometimes the compilation is deferred (placed on a queue for later compilation by the system at a "quiet time").

The impacts to be tested include the following:

- **Footprint**. This is a measurement of the physical file size of the components needed for the application as delivered.
- **Memory Use - Active.** Here, we define memory use as the Working Set of the application, tested in the absence of memory competition to prevent windows from arbitrarily reducing the working set size of the app.

- **Memory Use - Reuse.** Here, we consider shared versus private memory, as well as how where the memory is retained in the system. Again, we tested in a situation where plenty of memory was available to prevent paging to disk.
- **CPU Consumed.** To force all of the application to be loaded into ram, and also force any required JIT compilations to be performed on managed code, all lines of the test dll are executed in a single background thread. This is performed in the absence of competing applications (other than monitoring tools). Essentially, all lines of code of the dll are run serially to completion. The amount of CPU cycles consumed by the process to complete this is measured.
- **Time**. The amount of time consumed to run all of the lines of code (including any Jit compilation, if necessary) is also measured.

The impacts of CPU Consumed and Time are measured for first launch of the application.

Due to under-documented sizing limitations of Windows Universal Apps[1], the amount of code lines needed to be less that what was used in the other tests, to impact results are scaled based on the number of lines of test code for comparison purposes.

TEST APPLICATION(S)

The bad application that I developed to test with was simply a very large dll. Twenty years ago I wrote a 100MB dll called MassiveDll as part of an investigation of the impact of dll base address rebasing. The test app, called Massive Rebase, would load two dlls on demand and we could see what happens when an address conflict exists. Back then, it meant the memory sharing didn't occur for the rebased dll and every use consumed more memory. Since then Microsoft implemented Address Layout Space Randomization (ALSR) for security purposes, which inadvertently solved the rebasing issue by always performing rebasing once when the dll is loaded and then enabling the dll memory sharing.

Back then, on a 32-bit OS, a 100MB dll was big stufOf. I didn't want to borrow code, so instead I wrote a program to generate enough source code that I could build into a large dll. Today, we have a lot more memory so I dusted of the source code of the generator, modified a few inputs, and made a 500MB version of MassiveDll. That's 25 million lines of source code such as:

```
I += j%(k+1);
```

In the original version it was just a series of 5 lines like that repeated a lot. Because I was now interested in the new memory compression feature in Windows 10, I made the new code more random to mimic the compressibility of normal code, but basically the lines of code are like that shown above. This code was placed in methods of 500 lines each as part of a single class, plus a method to call all of the other methods in the class serially. The source code generator was also modified for an output option to produce a choice of either C# or C++ code.

[1] As of this writing, only limits for small platform project apps were documented. We were using much more than those limits, but were testing using WUA rather than Windows 8.1 or Windows Phone projects which has documented limits.

Minimal changes were made to the MassiveRebase app to load and test the dll. Primarily, I added a run button and timer to time the run of the code in the dll. Simply loading a large dll into memory was not enough for this testing, we needed to run all of the code in order to force the loading of pages into memory, Jit compilation (where needed), and measure the effectiveness of generated code.

I then created the C# versions similarly. This did not go as easily as anticipated. To begin with, the Visual Studio IDE is amazingly still a 32-bit process. I can count on the fingers of my left had the number of times I had seen Visual Studio itself crash prior to this work. An analysis of the event logs showed that I crashed it over 300 times this month trying to get all this code to build. With a lot of patience, I was able to coax out the two versions using WPF and .Net Framework 4.5.2.

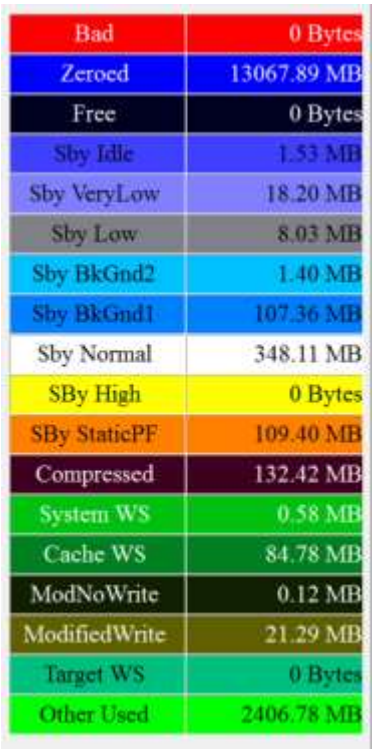
The Windows Universal App did not go as well. In the end, I was forced to cut down the size to 20% of the others, as that is about all I can get to build. The results in the paper are therefore presented in values of value per line of code so that comparisons may be made.

The design of the applications for Native and .NET applications allowed for loading of the application without loading the dll, and a button to load the dll on demand. The dynamic loading of the dll allowed for more precise numbers on the impact of the dll without impact of the application. Unfortunately, while reflection loading and referencing of classes/methods is supported in WUA, it does not appear possible to get the dll into a location where it may be dynamically loaded (this is part of the contract given by WUA to protect the system from unwanted code), so the dll is loaded into memory upon application launch instead.

All versions of software were built using the latest release version Visual Studio at the time of the testing, VS 2015 with SP1.

THE MONITORING TOOLS

For monitoring, my tool ATM provides a convenient graphical depiction of the memory use at a detailed level, and ultimately is useful to provide context regarding how the numbers changed over time. The tool was enhanced slightly to allow us to pull detail of a specific target application as well. The color code for the graph is shown on the right for reference. We will be interested in the Target Working Set (Teal), Modified Write List (Brown) and Standby Normal List (white).



Most of the monitoring numbers used to create the bar charts presented were produced by the test application itself, by ATM, or by Process Explorer which is good at producing raw numbers. Ultimately both tools are pulling numbers from the same kernel interfaces.

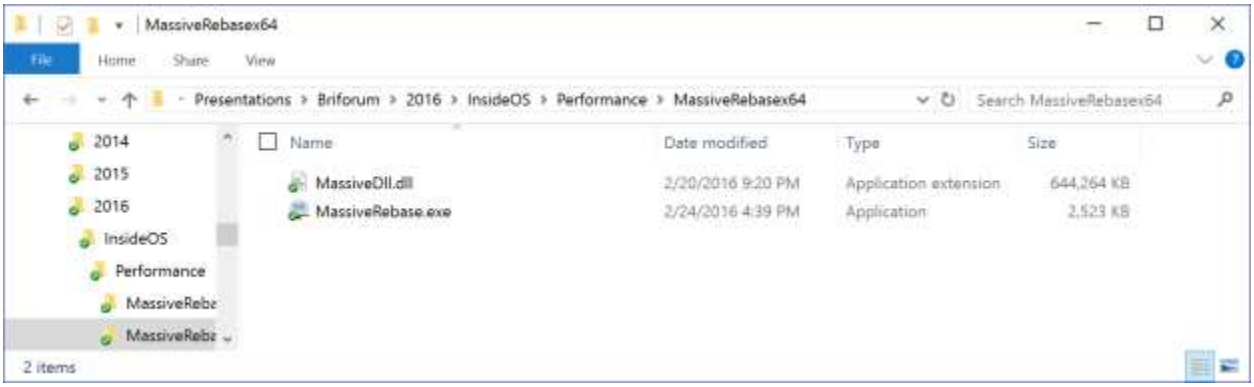
THE TEST SYSTEM

The test system was a Surface Book sporting a quad-core I7 processor, 16GB ram, and SSD disk subsystem running Windows 10 build 1511 (x64).

TEST RESULTS FOR "WIN32" C++

FOOTPRINT (WIN32)

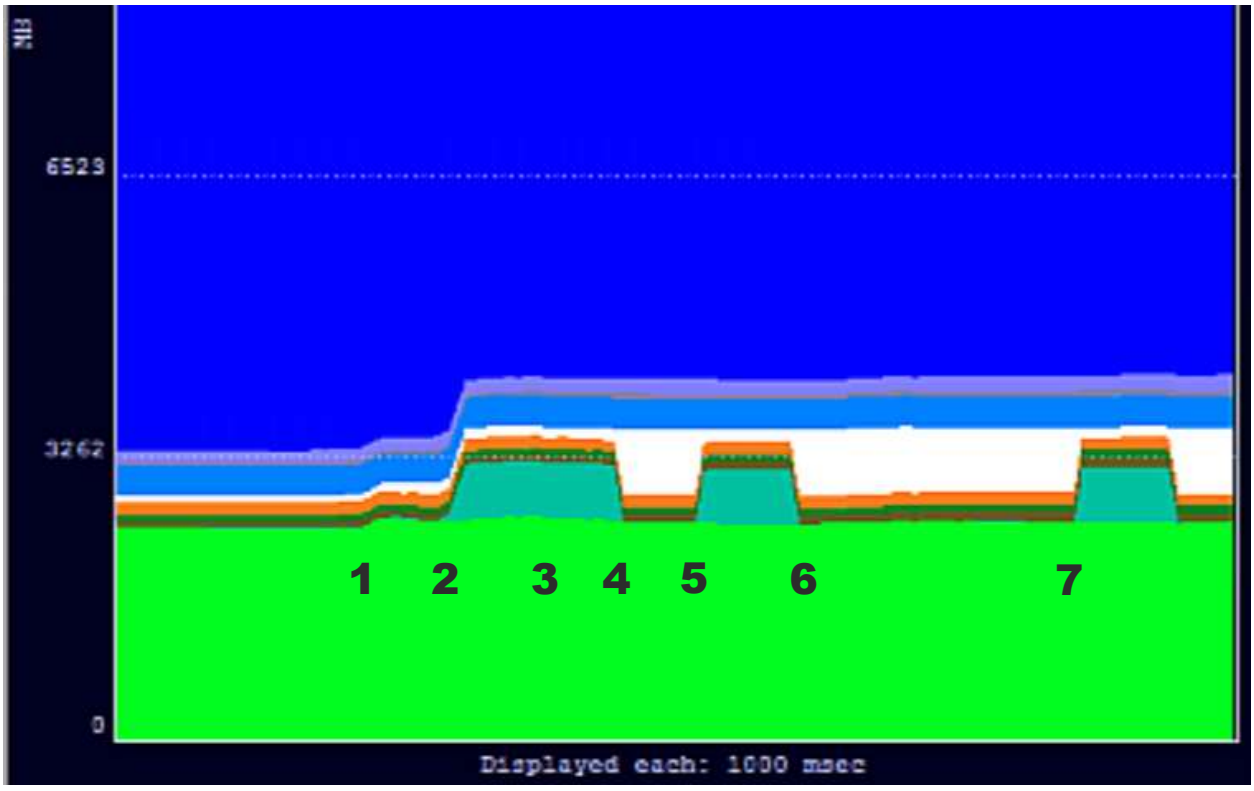
The components built for this project are shown in the image below. Both the executable and dll were built as 64-bit processes to be consistent with other tests. Some standard windows runtime libraries are also used by the process, but these would be present as part of the base operating system.



The dll created from the 25 million lines of code is 629MB, which works out to 26bytes per line of source code.

MEMORY (WIN32)

The Memory Trace taken by ATM is shown in the image below:



The events indicated on the image are:

Event 1: Target application launch. Loading of the dll is not marked on the image, but occurred about half way between events 1 and 2.
Event 2: Run of dll code to force all pages to be mapped into memory. In between events 2 and 3 (not marked on image) the dll code is run again to determine maximum possible speed when all code is in memory.
Event 3: Minimization of the Target application GUI
Event 4: Forced Trimming of the Target Application Working Set.
Event 5: Run of dll code to force page faults to bring all dll code back into working set.
Event 6: Application exit.
Event 7: Target application re-launch, dll load, and run.

MEMORY – LAUNCH AND INITIAL RUN (WIN32)

The working set of the process prior to dynamically loading the dll was just over 11.7MB.

Dynamically loading the dll, but not requesting processing, requires the windows cache manager to open the file and only page in small portions of the dll. After loading, the working set of the process was 12.7MB, an increase of about 1MB.

After running all of the code, the Working set grew by 625MB, similar to the size of the dll itself. Almost all of the memory added to the working set was marked "sharable" rather than private, an important distinction for the next category. From a binary efficiency, the memory space used is about 26 bytes per line of code.

MEMORY – MINIMIZED (WIN32)

Minimizing the application and bringing another application to the foreground, in the absence of an overall shortage of available memory, has no impact on the memory of the process, and all pages remain in the working set.

[Not shown in images]: If a second copy of the application is run at the same time, by this or another user on the same OS, that processes will also have a full working set, but only about 1MB is added to the total memory used on the system, indicating that the bulk of the dll code is being shared.

MEMORY – WORKING SET TRIMMING (WIN32)

On a system with sufficient RAM, trimming of the working space is unlikely. As was shown in Part 1 of this pair of white papers, running additional programs to consume remaining memory can trigger working set trimming. For these tests, we just used a special feature of ATM to request trimming of the target application on demand.

When trimming occurs, the system does not treat sharable memory pages the same way it treats private pages. Effectively this means code pages are treated differently than data in most cases.

Unlike the private "data" pages, we did not detect any memory compression used on code pages when trimmed.

As most of the memory associated with the Win32 style application/dll is sharable code pages, trimming is done without needing to ever page out to the page file (the original form exists on the disk already). So the OS just moves the management markings of the memory and we see the memory transfer from the working set of the target application and into the Standby List. As the processes is using normal priority memory, it is deposited into the "Normal" Standby List. It will remain in the standby list until needed, or until the OS requires more memory than can be satisfied by the zero and free pools (and lower priority standby lists).

For this application, including the exe and dll and all associated data, 639MB was trimmed and 630MB was added to the Normal Standby List.

MEMORY – RESTORAL OF WORKING SET (WIN32)

Restoral of the memory to the Working Set is accomplished by running the dll code again. As all of the code remained in the standby list without compression, access generates page faults in the kernel that are handled as "soft" page faults rather efficiently.

MEMORY – APPLICATION CLOSING (WIN32)

When all copies of the application are closed, the memory associated with those sharable code pages is released, but windows retains a copy in an appropriate Standby List.

MEMORY – SECOND USE (WIN32)

As long as the memory remains in the Standby List, an additional launch is quite simple and most of the memory will use existing pages from the Standby List.

CPU (WIN32)

CPU CONSUMED (WIN32)

A total of 2.95 billion CPU cycles were recorded as used by the application[2] during the initial run (Event 2). This averages out to be about 126 CPU cycles per line of code, which includes not only the code within the dll, but support by the OS to bring that code into memory.

For comparison, when the dll is run a second time while the code is in the active working set of the application, 706 million CPU cycles were needed (29.6 cycles per line of code).

[2] NOTE: Due to the way the code in the dll is organized, some additional CPU usage occurred as part of the memory mapping performed by the windows file system cache performing "read ahead" operations. No attempt was made to track that CPU and it is being ignored in these tests, but we see that about 500 read-ahead operations were performed (this number would have been higher had the disk not been an SSD).

TIME TO EXECUTE (WIN32)

The time to execute is measured by the application itself. The time to execute this code in Event 2 was 2.951 seconds, or roughly 118ms per million lines of code.

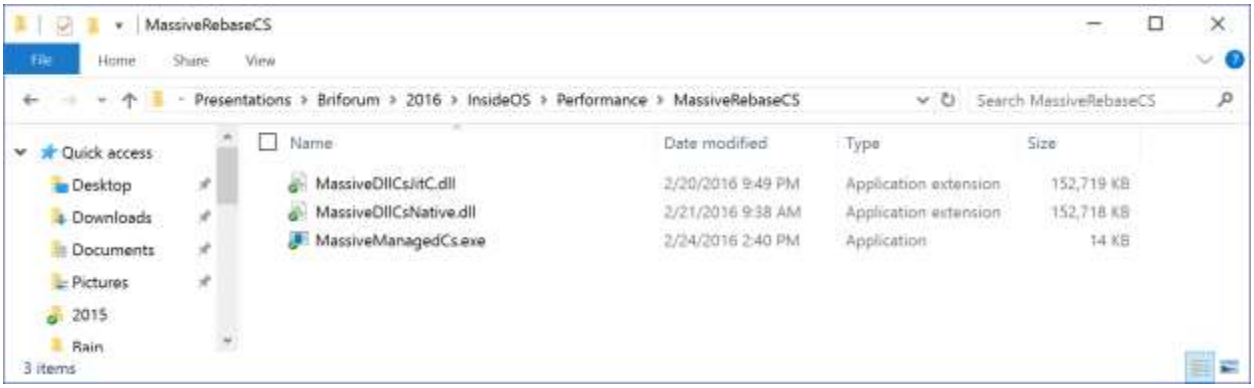
The time to execute is not just a function of the CPU cycles captured above, but other activities may be required that prevent the code from executing.

In this test, the pages of code were needed to be read into RAM from Disk by the system file cache manager whenever a page fault occurred as the process attempted to use code pages not yet in memory. The Windows cache manager does use a read-ahead strategy which kicked in during this test rather significantly, due to the serial nature of the way the code in MassiveDll is organized. This means that most of the time the cache manager has already at least started reading in the required page before the page fault occurred, eliminating much of the potential delays. Real world applications would probably not be quite as lucky and the time to execute code in a more random order would have resulted in a slightly higher average than recorded here.

TEST RESULTS FOR MANAGED C# .NET IL (JIT COMPILED)

FOOTPRINT (.NET JIT)

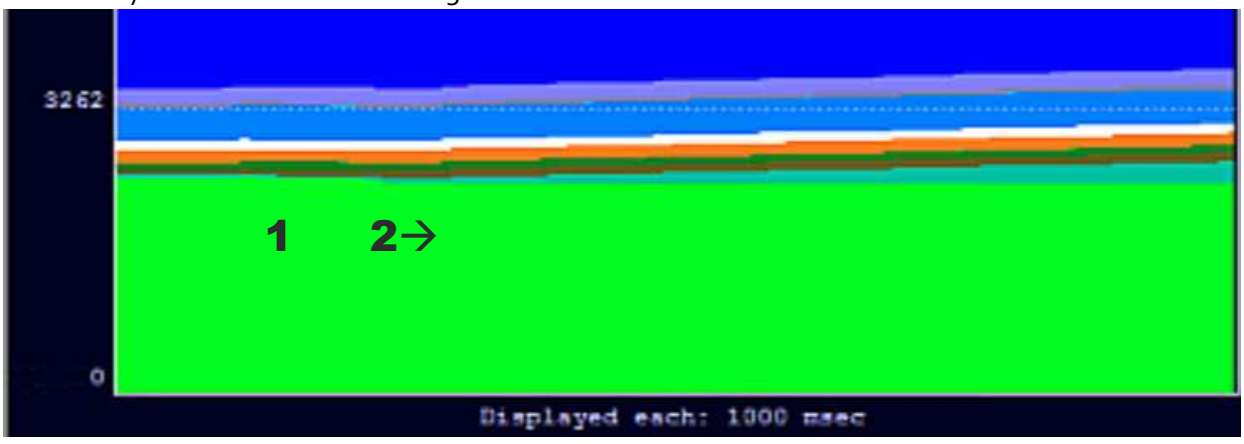
The image below shows the distribution folder for the IL version of Massive App and two copies of the DLL. Only the exe and the MassiveDllCsJit.dll files are used in this test (the other file is used in the test in the section that follows).



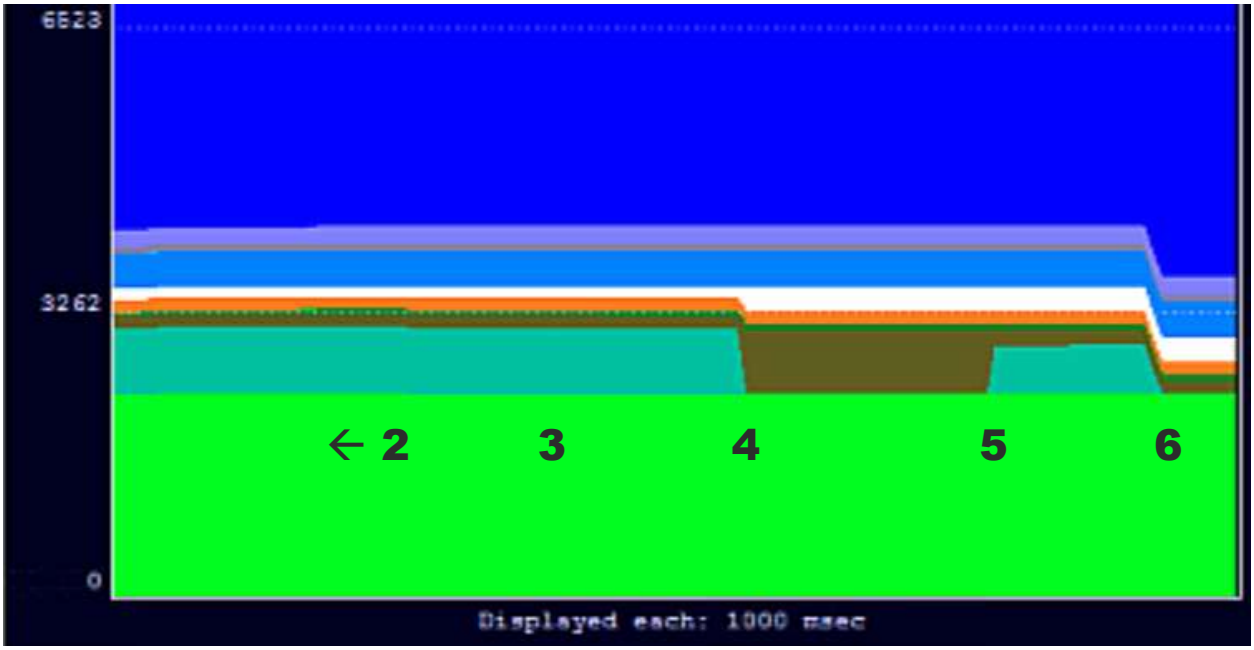
The compiled version of the dll contains Microsoft Intermediate Language (MSIL) code. The MSIL is a bytecode representation of the source code, but not something that can be executed by a CPU directly. Notice that the size of the DLL here (149MB) in comparison to the C++ compiled dll (629MB) in the prior test. By contrast, the source code was about 350MB. Clearly MSIL is a more efficient means to transfer the program to the target computer.

MEMORY (.NET JIT)

Due to the need to just-in-time compile the IL code, event 2 takes a very long time. The beginning of the memory trace is shown in the image below.



The event ids are consistent with the original example. The relevant portion of the memory timeline is shown in the next image:



MEMORY – LAUNCH AND INITIAL RUN (.NET JIT)

The working set of the process prior to dynamically loading the dll was just over 41.9MB. This is significantly larger for a base application that actually has fewer features (as there was additional functions in the main app of the Win32 that I left in unused and did not duplicate in the newer test apps).

Dynamically loading the dll, but not requesting processing, requires the windows cache manager to open the file and only page in small portions of the dll. After loading, the working set of the process was 42.9MB, an increase of about 1MB.

After running all of the code, the Working set grew to 764MB. During that time, the IL code was "Just In Time Compiled", with the JIT counter for the process increased by 50115, which is very close to the number of methods contained in the 25 million lines of code. , Most of the added memory added to the working set was marked "private". Analysis seems to show that the IL bytecode pages are sharable, but the compiled code is marked private.

MEMORY – MINIMIZED (.NET JIT)

Minimizing the application and bringing another application to the foreground, in the absence of an overall shortage of available memory, has no impact on the memory of the process, and all pages remain in the working set.

If a second copy of the application is run at the same time, the JIT compilation starts again, and the compiled form is again placed in different private memory. (Not shown in images).

MEMORY – WORKING SET TRIMMING (.NET JIT)

When trimming of the working set is performed, the original IL code is moved to the Standby list, while the compiled code that is in private memory is sent to the Modified Write queue. In the absence of memory pressure, it remains in the queue without writing to the page file.

Of the 765MB working set in place prior to trimming, 153MB went to the standby list (representing the exe application and IL code of the dll) and 579MB went to the Modified Write Queue. This leaves 28MB unaccounted for, but I suspect there may have been data that was freed by the garbage collector.

MEMORY – RESTORAL OF WORKING SET (.NET JIT)

The restoral of the working set must bring in code from the exe that is needed and the compiled version of the dll code private memory. There is no need to restore the IL code so it remains in the standby list.

The working set grew by a total of only 540MB, with 534 coming from the Modified Write queue and 1MB from the Standby List.

MEMORY – APPLICATION CLOSING (.NET JIT)

When all copies of the application are closed, code from the exe and the IL code remain in the standby list.

MEMORY – SECOND USE (.NET JIT)

If the application is restarted, the code that remains in the Standby list is used, but the JIT compilation has to run again.

CPU (.NET JIT)

CPU CONSUMED (.NET JIT)

A total of 866 billion CPU cycles were recorded as used during the initial run! This averages out to be about 37,230CPU cycles per line of code. As we will see in the following test (when native compilation is used), this is almost completely due to the code being in IL bytecode format and not yet compiled. The JIT compilation, which occurs in process, is responsible for most of these CPU Cycles.

It is important to note that without NGEN requested compilation, this JIT Compilation will occur every time the process is started and run. The compilation results are not saved to disk and will be discarded when the process exits.

TIME TO EXECUTE (.NET JIT)

Thanks to the time spent in Jit, the time to execute this code in the initial run was 5 minutes 53 seconds, or roughly 13,704ms per million lines of code.

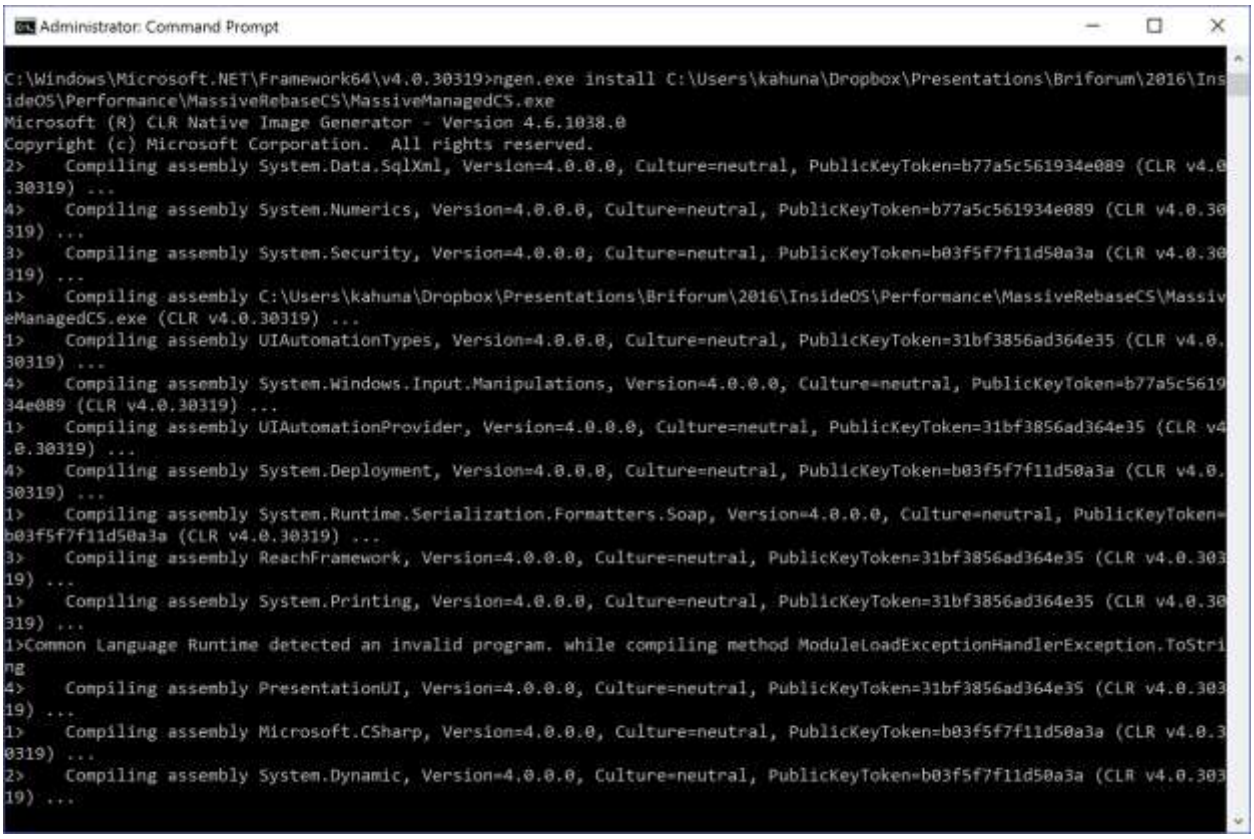
A secondary run of the dll from the Jited code took 0.33 seconds, which is actually faster than the non-managed Win32 app.

TEST RESULTS FOR MANAGED C# WITH PRIOR NATIVE COMPILATION

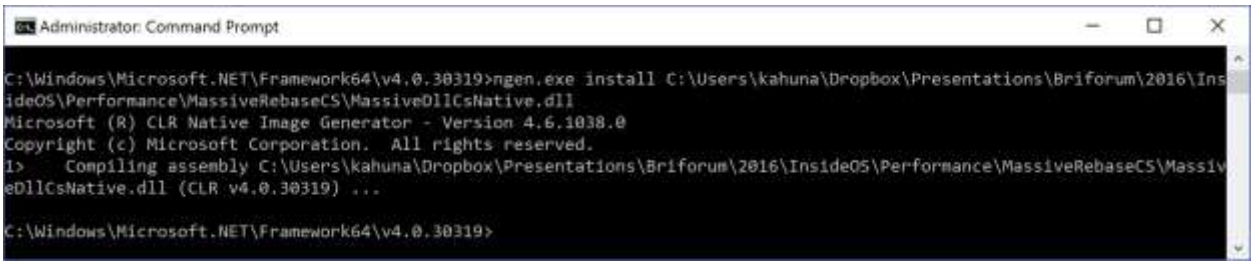
FOOTPRINT (.NET NATIVE)

In addition to the IL files as shown in the previous example, NGEN compilation at the target client is needed. This is typically done by an installer, but we can do so by hand.

Natively compiling these two files involved running the ngen.exe command with the install option against the exe and dll files shown in the prior section[3].

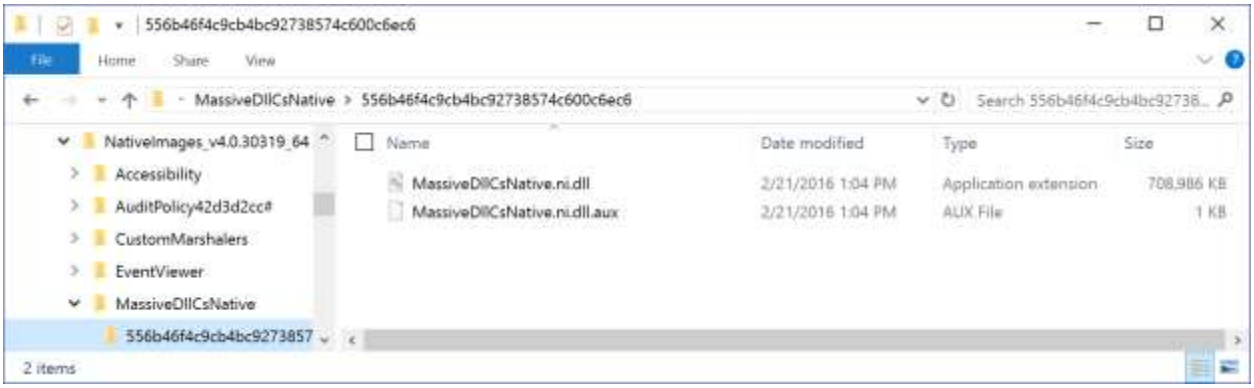


Compiling the dll is done similarly.



[3] To aid in comparisons, the exe was actually natively compiled prior to performing the tests run in the previous section of this paper, so that only the test dll required JIT compilation.

This generates additional "native compiled" versions that we can find in the Windows Assembly:



We still need the uncompiled versions of the program files, as these will be the target of shortcuts and the actual load request. The OS will then locate the NI versions in the assembly cache and use them for the processing instead.

The size of the compiled NI dll is about 7.5% larger than the C++ equivalent, but if you include the IL copy it represents about a 30% increase in the footprint on the target disk.

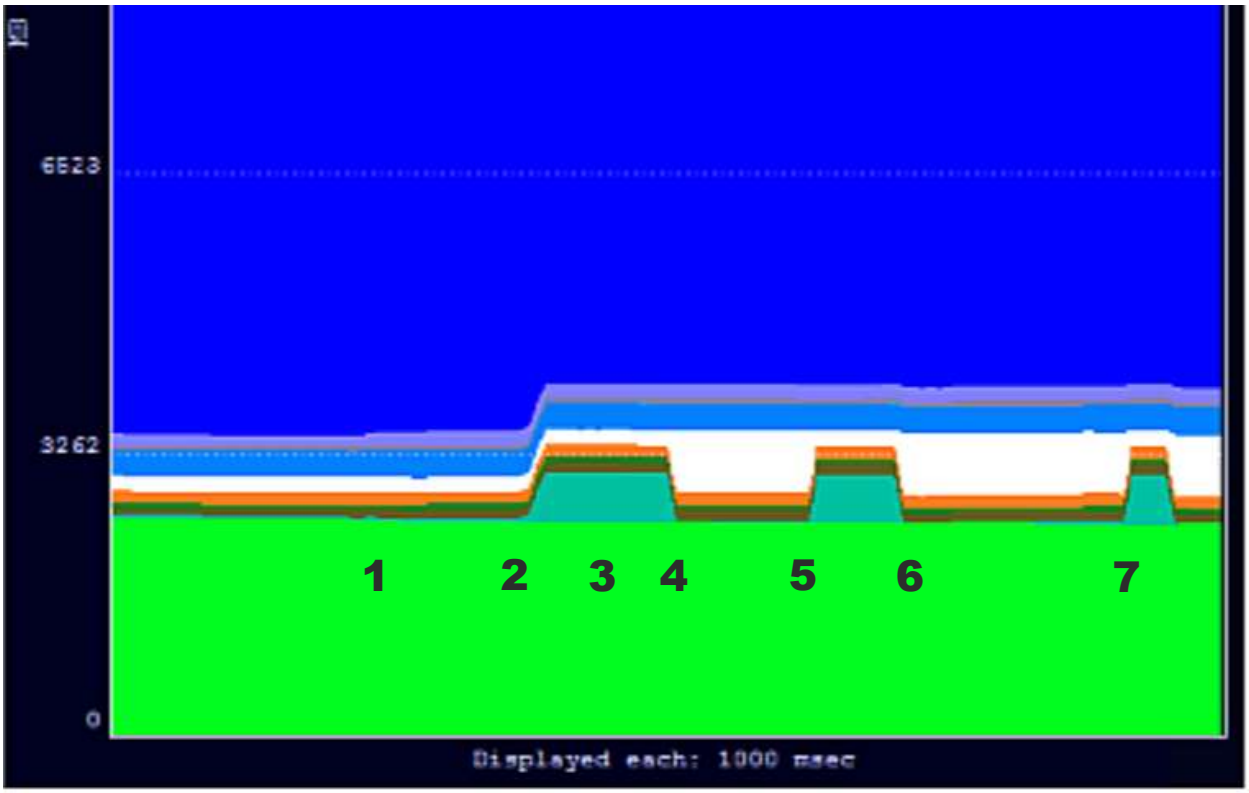
When .Net first came out, NGEN compilation was different on different processors, for example Intel and AMD x86 processors. So the approach was always to compile upon installation at the target device. Over time, Microsoft has eased this limitation[4] so that an x86 compilation is identical when done not only on Intel and AMD x86 processors, but when compiled for x86 on the x64 versions (when the x86 version of NGEN performs the compilation). The compilation is still different on processors of other architectures, such as ARM. And the compilation is different if the source is AnyCPU and it is compiled into 64bit by the 64bit NGEN. With an AnyCPU program, on x64 it will run as x64 so to get the benefits of native compile you would have to distribute the x64 NI for those systems. As this is way too confusing, Microsoft just kept mum about pre-compilation as part of the distribution process and the practice of compiling on the target device as part of the installation continues today.

Unfortunately, not every installer asks for compilation. Famously, even Microsoft forgot to ask for compilation of the entire .Net Framework in one release! On top of this, many administrators forget that after installation of software it is necessary to let the system sit to allow the background compilation queue to be cleared before Sysprepping their image (or closing monitoring mode in the App-V Sequencer or other application repackaging tools). Which is why we are running this test without compilation in the previous section and with compilation here.

[4] It was necessary for Microsoft to fix this to enable image distribution management, where a company creates a golden image of the installed Windows OS partition with applications installed and SysPrepped, and distributes this image to various hardware and virtual machines.

MEMORY (.NET NATIVE)

The memory trace for this example is shown in the image below. The events follow the same number scheme.



The general memory use in this case is much more similar to that of the "Win32" application.

LAUNCH AND INITIAL RUN (.NET NATIVE)

The working set of the process prior to dynamically loading the dll, and after, was very close to the uncompiled scenario presented previously.

After running all of the code, the Working set grew by only 550MB, as the IL code was not required. The Jit counter increased by only 14, indicating that the natively compiled version was used. Verification of the native dll being used is also provided by the modules view of Process Explorer. The added memory to the working space was primarily marked as sharable.

MEMORY – MINIMIZED (.NET NATIVE)

Minimizing the application and bringing another application to the foreground, in the absence of an overall shortage of available memory, has no impact on the memory of the process, and all pages remain in the working set.

If a second copy of the application is run at the same time, by this or another user on the same OS, that processes will also have a full working set, but only about 3MB is added to the total memory used on the system, indicating that the bulk of the dll code is being shared.

MEMORY – WORKING SET TRIMMING (.NET NATIVE)

When trimming of the Working Set of the target process occurs, 591MB was removed. 538MB was moved to the Standby List and 23MB to the Modified Page Queue, leaving about 30MB presumably freed by Garbage Collection.

MEMORY – RESTORAL OF WORKING SET (.NET NATIVE)

Of the 547MB restored, 532 came from the Standby List.

MEMORY – APPLICATION CLOSING (.NET NATIVE)

When all copies of the application are closed, the memory associated with those sharable code pages is released, but windows retains a copy in an appropriate Standby List for potential reuse.

MEMORY – SECOND USE (.NET NATIVE)

If the application is restarted, the code that remains in the Standby list is used.

CPU (.NET NATIVE)

CPU CONSUMED (.NET NATIVE)

A total of 2.72 billion CPU cycles were recorded as used during the initial run. This averages out to be about 117CPU cycles per line of code, which is about 8% more efficient than the unmanaged Win32 app!

TIME TO EXECUTE (.NET NATIVE)

The time to execute this code for initial run was 2 seconds, or roughly 80ms per million lines of code. This was actually much faster than the unmanaged code in the code system testing performed, a 47% improvement in this rather unusual test[5].

[5] As for real-world applications, it is as they say in the car commercials -- your mileage may vary.

TEST RESULTS FOR WINDOWS UNIVERSAL APP C# WITHOUT SUSPENDING

Due to limitations in Visual Studio for building Universal Apps, and the design criteria for Universal Apps in general, the WUA app is constructed slightly differently:

- The dynamic dll loading cannot be effectively done, so the dll is loaded upon startup of the application.
- I had to limit the size of the dll to fit in the package. The example uses a 1/5 size dll (5 million lines of code instead of 25).

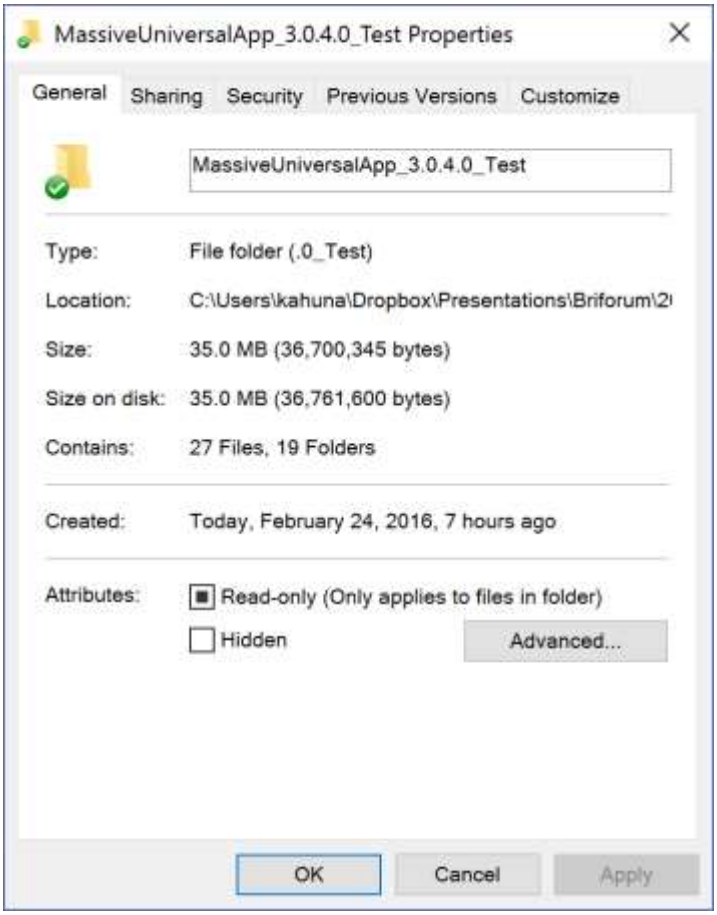
The OS also works differently depending on the state of the application. WUA apps may become suspended, a state where foreground threads are frozen because the user is not actively using the application and the OS has specific behaviors that are different when the app is suspended.

The tests of the Windows Universal App are run in this section, without suspending the application, and then again in the next section using suspension.

FOOTPRINT (WUA WITH OR WITHOUT SUSPENSION)

The MassiveDll that was built as a Universal App Dll for the package was 29.8MB in size when created.

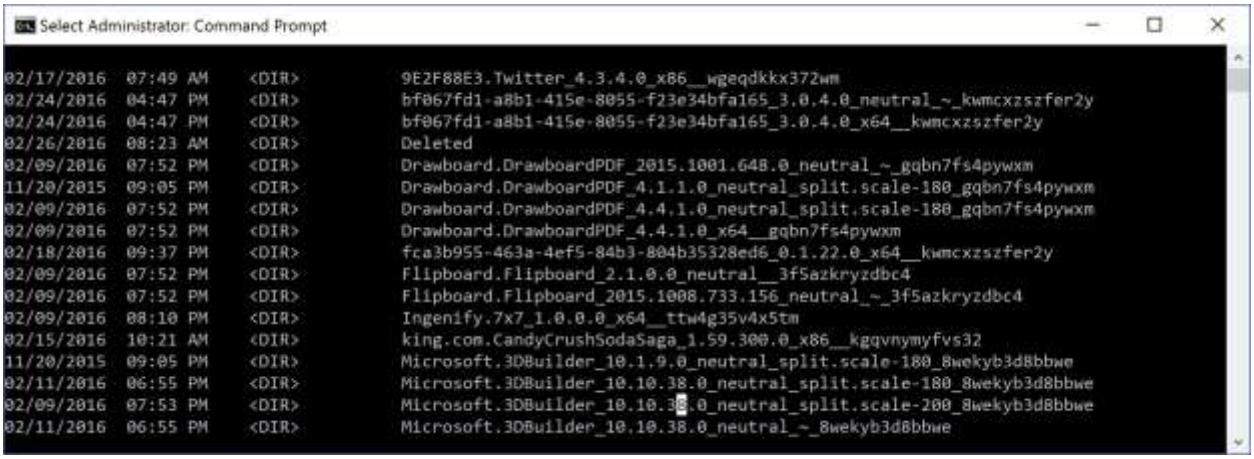
The size of the entire packaged app, which would be loaded into the Windows Store or Side-Loaded, is shown in the image below:



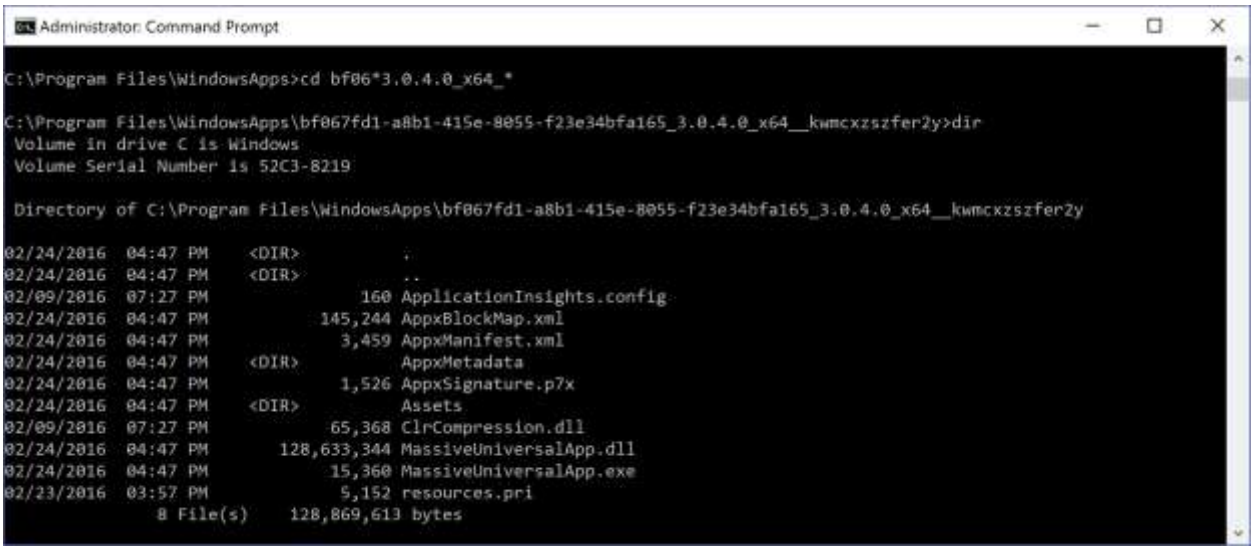
However, looking at sizes may not be that simple…

Windows Universal App Packages create a single native compiled dll that includes this dll and all other dependent dlls, including much of the .NET RT runtime. Microsoft refers to this as the ToolChain. The uncompiled version of this combination dll was 122MB, so obviously the dependencies were quite large prior to compilation. As we see when the code is run, however, this makes for very efficient processing.

When a Windows Universal App is installed into a Windows 10 Operating System, it is placed in a new folder under a hidden and protected folder "C:\Program Files\WindowsApps". We can see this folder by using an elevated cmd window and changing to the directory. The folder for our test application (as well as a prior test version) can be seen in the following image.



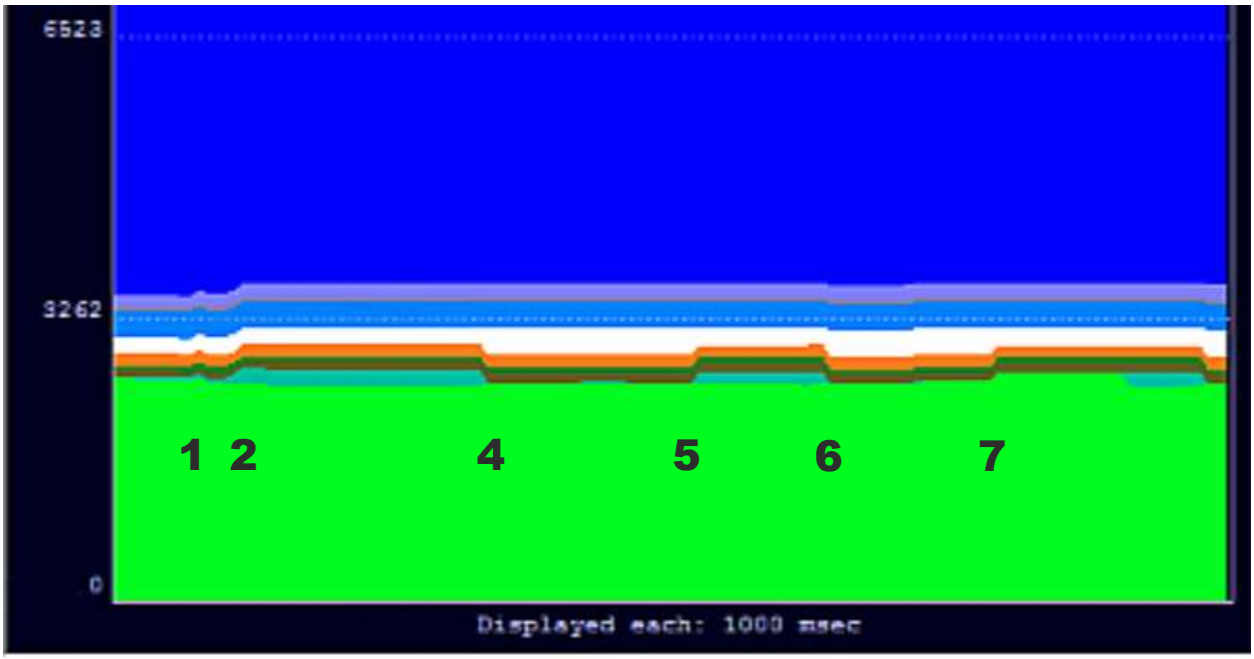
Our test program directory may then be displayed as shown here:



So our installed footprint, compiled with the entire tool chain required for the program comes in at 122.7MB (obviously file compression is being used in the delivery mechanism, but we didn't create MSI installers for the prior examples which also would have compressed the components).

MEMORY (WUA, NOT SUSPENDED)

The memory usage patterns for the Windows Universal app when not suspended are shown in the image below.



MEMORY – LAUNCH AND INITIAL RUN (WUA, NOT SUSPENDED)

The working set of the process with the dll loaded prior to run was larger than for the NGEN compiled.Net app (51MB versus 42MB). This may be because WUA builds the entire toolchain, meaning the dll and all dependencies, into a single dll that is loaded.

After running all of the code, the Working set grew to only 171MB, and the Jit counter did not increase, indicating that the entire toolchain compiled version was used. The added memory to the working space was primarily marked as sharable. This is considerably smaller than even the native .net app which grew by 592MB.

MEMORY – MINIMIZED (WUA, NOT SUSPENDED)

Minimization (Event 3) was not performed on this test scenario.

MEMORY – WORKING SET TRIMMING (WUA, NOT SUSPENDED)

When the working set of the target application was trimmed, 170MB was removed from the working set. 135MB were added to the Standby Normal list, and 7MB was added to the Modified Write queue. As is the case for WPF, there was a remaining 28MB that was not accounted for. There are some small differences in a number of queues, notably Standby Static, Standby Background 1, lower priority Standyby lists, and the File System Cache, but both memory compression working space and page file use remained static.

MEMORY – RESTORAL OF WORKING SET (WUA, NOT SUSPENDED)

When restoral of the working set of the target application is triggered by running the code in the dll, most of the working set returns, increasing by 131 MB to a total of 139MB. (122MB came from the Standby Normal and Modified Write Queue). Portions of the app outside the dll that are not needed would remain trimmed, so this is expected behavior.

Note that it is not possible (at this time) for a user to have two copies of the same app running similtaniously, so simultanoius app usage was not looked at[6].

MEMORY – APPLICATION CLOSING (WUA, NOT SUSPENDED)

When all copies of the application are closed, the memory associated with those sharable code pages is released, but windows retains a copy in an appropriate Standby List for potential reuse.

MEMORY – SECOND USE (WUA, NOT SUSPENDED)

Starting the application a second time makes effective use of the code in the standby list.

[Note: in the memory trace image, at event 7 you can see the "Other used" memory increase, and then a while later the target application shows the same memory. This is just an artifact of the ATM tool in which must I must manually select the target application after launch (and I forgot to do so for a while). The increase in other used memory at event 7 is the usage by the target app.]

[6] This is probably testable with an identical copy of the app, or under Server 2016 RDS. As Server 2016 RDS is not yet in release, this was not tested for this paper.

CPU (WUA, NOT SUSPENDED)

ABOUT SUSPENSION…

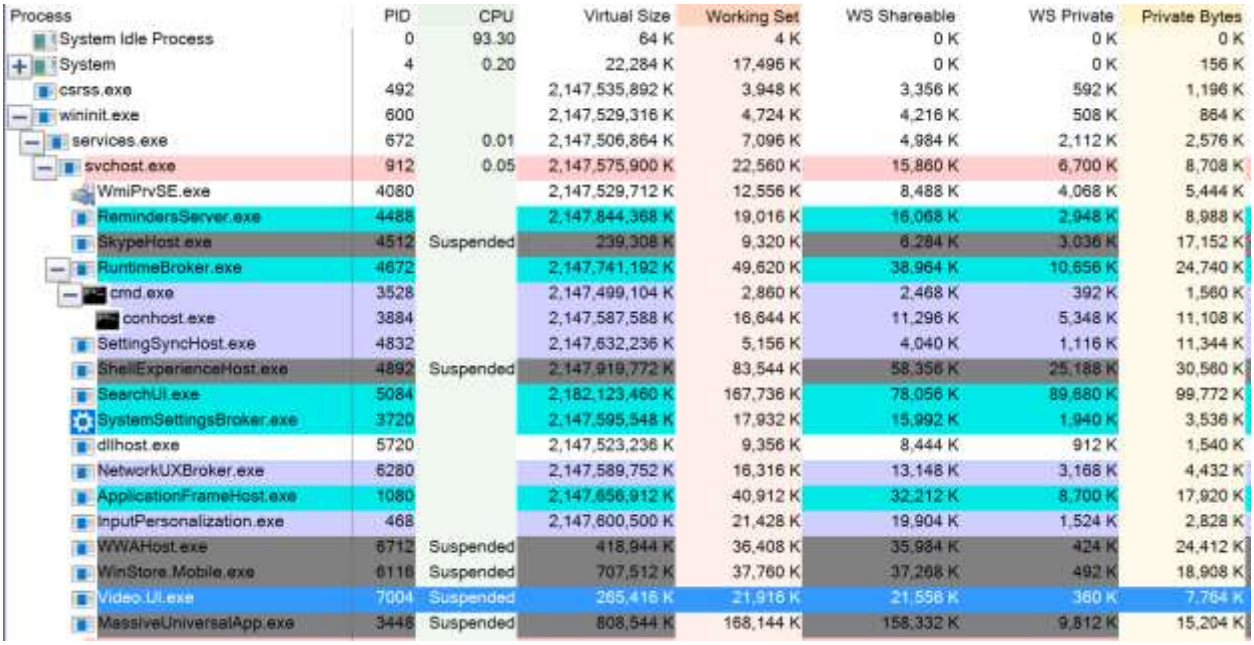
Universal apps are generally designed to remain running, and design guidelines heavily discourage the adding of exit buttons. When in tablet mode on Windows 10, or when running in the modern side 8/8.1 there also is no window title bar area for an exit ("the X button), however, in the more typical desktop mode of Windows 10, the user now has that option. Perhaps this will lead to a change in the design guidelines in the future, but for now they are discouraged.

Microsoft's current approach with WUA apps at present is to let the apps keep running, but suspend them. Initially, upon suspension, the operating system would remove move private, but not the sharable, memory from the working set of the application. This was changed in the Redstone build of Windows 10 and this trimming is no longer performed due to suspension. It is possible that suspension may increase the odds of memory trimming if the zero and free memory of the OS goes too low, but this had not been tested. Currently, suspension is all about CPU.

We can see the state of WUA apps in the CPU column of Process Explorer. The image below shows a few suspended apps, but MassiveUniversalApp.exe (at the bottom of the list) is not suspended. At this point the window was visible, but I had selected another window for active user interface input.

Process	PID	CPU	Virtual Size	Working Set	WS Shareable	WS Private	Private Bytes
System Idle Process	0	87.72	64 K	4 K	0 K	0 K	0 K
System	4	1.06	22,284 K	17,496 K	0 K	0 K	156 K
csrss.exe	492		2,147,535,892 K	3,948 K	3,356 K	592 K	1,196 K
wininit.exe	600		2,147,529,316 K	4,724 K	4,216 K	508 K	864 K
services.exe	672	0.01	2,147,506,864 K	7,092 K	4,984 K	2,108 K	2,572 K
svchost.exe	912	0.04	2,147,575,900 K	22,564 K	15,860 K	6,704 K	8,700 K
WmiPrvSE.exe	4080	0.07	2,147,529,712 K	12,556 K	8,488 K	4,068 K	5,444 K
RemindersServer.exe	4488		2,147,843,856 K	18,984 K	16,068 K	2,916 K	8,904 K
SkypeHost.exe	4512	Suspended	239,308 K	10,416 K	7,008 K	3,408 K	17,152 K
RuntimeBroker.exe	4672	< 0.01	2,147,737,568 K	49,460 K	38,756 K	10,704 K	24,738 K
cmd.exe	3528		2,147,499,104 K	2,860 K	2,468 K	392 K	1,560 K
conhost.exe	3884		2,147,587,588 K	16,644 K	11,296 K	5,348 K	11,108 K
SettingSyncHost.exe	4832	< 0.01	2,147,632,236 K	5,160 K	4,044 K	1,116 K	11,344 K
ShellExperienceHost.exe	4892	Suspended	2,147,919,772 K	83,560 K	58,372 K	25,188 K	30,560 K
SearchUI.exe	5084	0.01	2,182,127,556 K	168,004 K	78,052 K	89,952 K	100,096 K
SystemSettingsBroker.exe	3720	< 0.01	2,147,595,548 K	17,932 K	15,992 K	1,940 K	3,536 K
dllhost.exe	5720		2,147,523,236 K	9,356 K	8,444 K	912 K	1,540 K
NetworkUXBroker.exe	6280		2,147,589,752 K	16,276 K	13,148 K	3,128 K	4,388 K
ApplicationFrameHost.exe	1080	0.22	2,147,657,104 K	40,868 K	32,368 K	8,500 K	17,840 K
InputPersonalization.exe	468	< 0.01	2,147,600,500 K	21,428 K	19,904 K	1,524 K	2,828 K
WWAHost.exe	6712	Suspended	418,944 K	36,408 K	35,984 K	424 K	24,412 K
WinStore.Mobile.exe	6116	Suspended	707,512 K	37,760 K	37,268 K	492 K	18,908 K
Video.UI.exe	7004	Suspended	265,416 K	21,916 K	21,556 K	360 K	7,764 K
MassiveUniversalApp.exe	3448	0.07	815,624 K	169,064 K	157,972 K	11,092 K	16,580 K

After minimizing the app window, the app is suspended after a second or so. Typically, WUA apps are designed to trigger on the initiation of suspension, allowing them to save off state data prior to suspension.



Process	PID	CPU	Virtual Size	Working Set	WS Shareable	WS Private	Private Bytes
System Idle Process	0	93.30	64 K	4 K	0 K	0 K	0 K
System	4	0.20	22,284 K	17,496 K	0 K	0 K	156 K
csrss.exe	492		2,147,535,892 K	3,948 K	3,356 K	592 K	1,196 K
wininit.exe	600		2,147,529,316 K	4,724 K	4,216 K	508 K	864 K
services.exe	672	0.01	2,147,506,864 K	7,096 K	4,984 K	2,112 K	2,576 K
svchost.exe	912	0.05	2,147,575,900 K	22,560 K	15,860 K	6,700 K	8,708 K
WmiPrvSE.exe	4080		2,147,529,712 K	12,556 K	8,488 K	4,068 K	5,444 K
RemindersServer.exe	4488		2,147,844,368 K	19,016 K	16,068 K	2,948 K	8,988 K
SkypeHost.exe	4512	Suspended	239,308 K	9,320 K	6,284 K	3,036 K	17,152 K
RuntimeBroker.exe	4672		2,147,741,192 K	49,620 K	38,964 K	10,656 K	24,740 K
cmd.exe	3528		2,147,499,104 K	2,860 K	2,468 K	392 K	1,560 K
conhost.exe	3884		2,147,587,588 K	16,644 K	11,296 K	5,348 K	11,108 K
SettingSyncHost.exe	4832		2,147,632,236 K	5,156 K	4,040 K	1,116 K	11,344 K
ShellExperienceHost.exe	4892	Suspended	2,147,919,772 K	83,544 K	58,356 K	25,188 K	30,560 K
SearchUI.exe	5084		2,182,123,460 K	167,736 K	78,056 K	89,680 K	99,772 K
SystemSettingsBroker.exe	3720		2,147,595,548 K	17,932 K	15,992 K	1,940 K	3,536 K
dllhost.exe	5720		2,147,523,236 K	9,356 K	8,444 K	912 K	1,540 K
NetworkUXBroker.exe	6280		2,147,589,752 K	16,316 K	13,148 K	3,168 K	4,432 K
ApplicationFrameHost.exe	1080		2,147,656,912 K	40,912 K	32,212 K	8,700 K	17,920 K
InputPersonalization.exe	468		2,147,600,500 K	21,428 K	19,904 K	1,524 K	2,828 K
WWAHost.exe	6712	Suspended	418,944 K	36,408 K	35,984 K	424 K	24,412 K
WinStore.Mobile.exe	6116	Suspended	707,512 K	37,760 K	37,268 K	492 K	18,908 K
Video.UI.exe	7004	Suspended	265,416 K	21,916 K	21,556 K	360 K	7,764 K
MassiveUniversalApp.exe	3448	Suspended	808,544 K	168,144 K	158,332 K	9,812 K	15,204 K

CPU CONSUMED (WUA, NOT SUSPENDED)

A total of 2.3 billion CPU cycles were recorded as used during the run. This averages out to be about 207CPU cycles per line of code, which is significantly higher than any prior scenario. Some of that average increase would be due to a fixed overhead of loading code regardless of size, and with a smaller payload the increase is suspected. I did not attempt to redo the prior tests with the same payload size to confirm this theory, mostly because the reality is that WUA apps will be smaller making these test better reflect field reality.

TIME TO EXECUTE (WUA, NOT SUSPENDED)

The time to execute this code was 1.54 seconds, or roughly 308ms per million lines of code. Like the CPU consumed result, at least a portion of the per lines of code average is high due to the smaller package size.

This was actually faster than the unmanaged code in the code system testing performed. I believe these differences would fade on warmed up systems where other software using shared libraries were previously run.

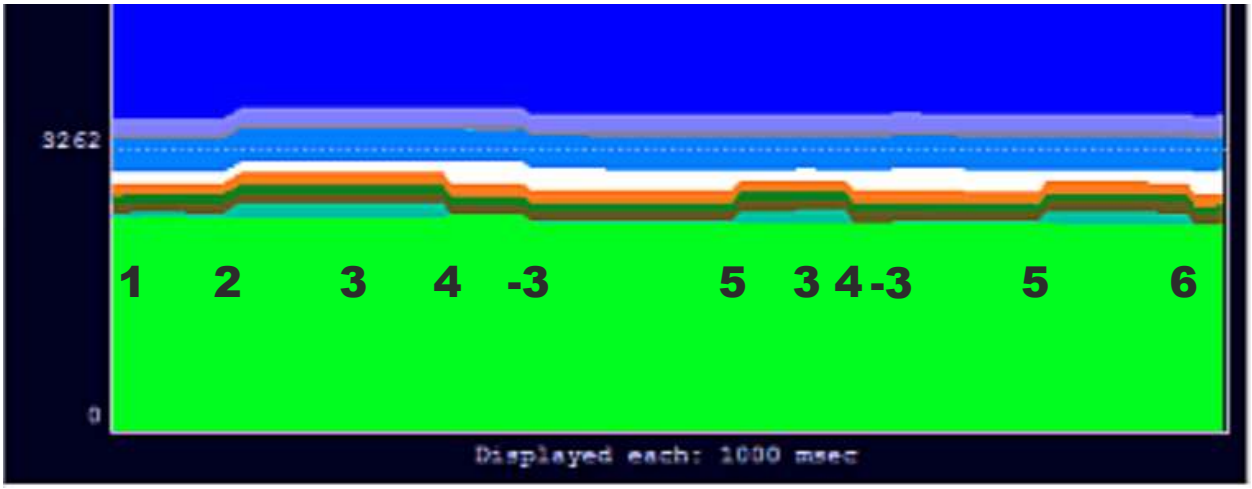
Tests of the .NET Native app and the WUA app when warmed up should eliminate most of the size difference issues. Those test showed an execution time minimum of 9.5 versus 12.8 ms per million lines of code, indicating that the WUA apps are about 26% less efficient at code execution. It is likely that this is something that can be overcome by Microsoft over time.

TEST RESULTS FOR WINDOWS UNIVERSAL APP C# WITH SUSPENSION

Only differences in the previous scenario are covered here.

MEMORY (WUA, WITH SUSPENSION)

The memory usage patterns for the Windows Universal app with suspension are shown in the image below.



The "-3" event is the un-suspending of the target application by bringing the UI back to the foreground.

Overall, suspension has little effect on memory consumption in the Redstone release (which is a change from Threshold). Trimmed memory is treated about the same when trimming as was the non-suspended case.

I considered removing this section from the paper after the latest OS changes, however a curiosity occurs when the application is unsuspended at the new "-3" events. The cause of the decrease in memory consumption is left for future study.

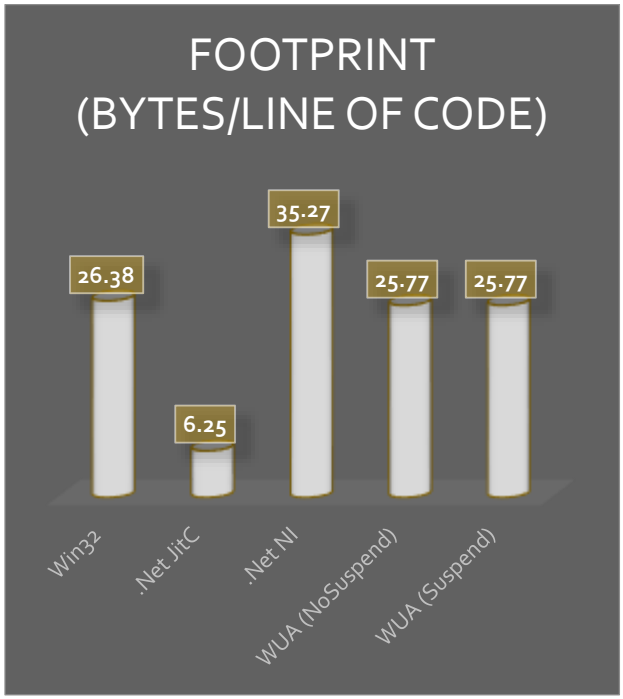
The inclusion of this section in the summary graphs at the end of the paper also provide an indication of the "margin of error" in the measurements made, as the two WUA results should be pretty much identical on those measurements.

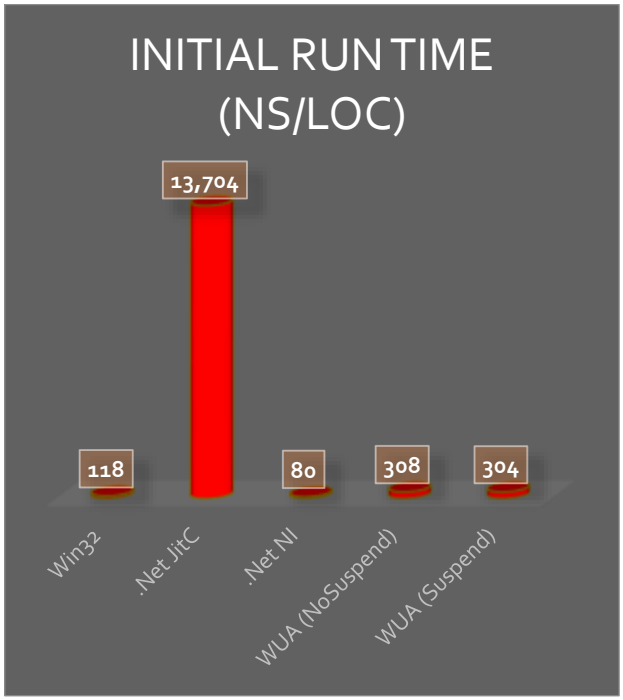
CPU (WUA, WITH SUSPENSION)

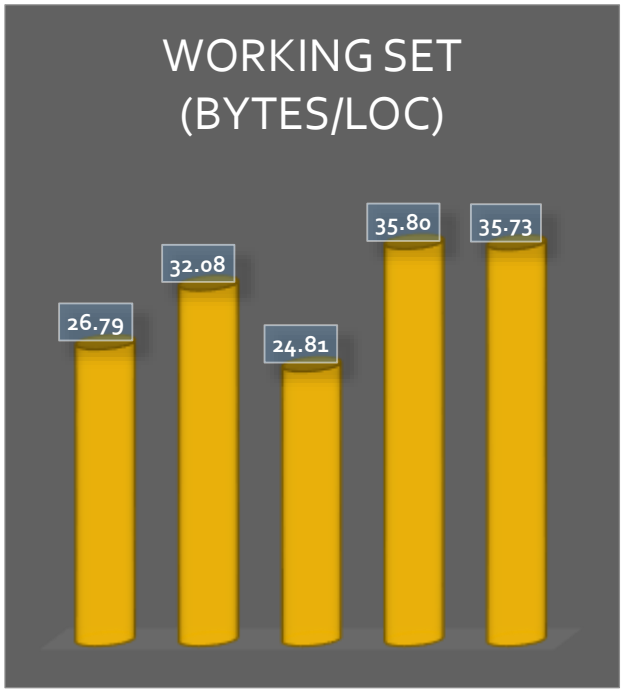
No significant differences detected, other than the foreground threads of the target application not consuming CPU while suspended.

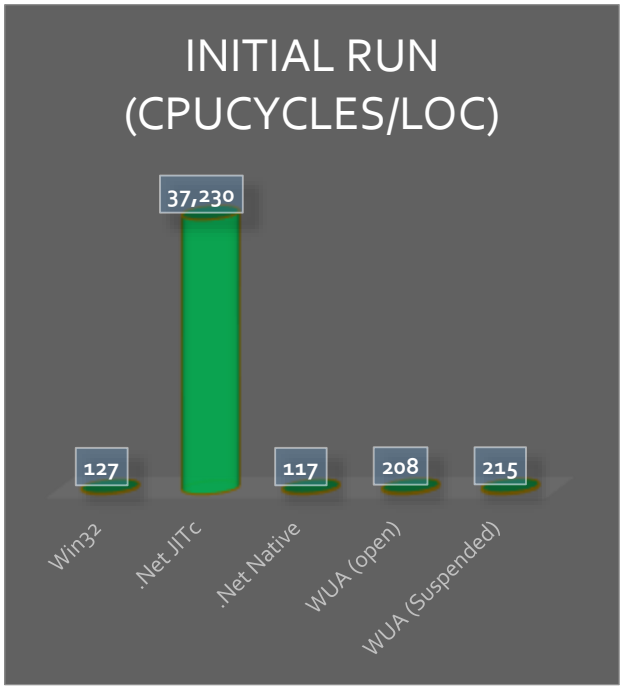
SUMMARY RESULTS AND INTERPRETATION

The charts below illustrate the key performance measurements made in the testing:









We can clearly see the importance of NGEN compilation of managed software in these results. Although IL is a nice compact form for delivery, the impact of sometimes not having it compiled has the most severe impact. In addition to setup programs that forget to compile, or situations where background NGEN hasn't completed, this problem exists due to developers not keeping up with their dependencies of their build project. In the example of one vendor, they included version 2.0 of some Google API in the project references, but were linking with version 4.0. This reeks havoc with JIT or NGEN compilation; eventually it gets compiled with version 4 of the dependency, but it can add up to 5 seconds to the compilation! When the application was used in App-V, this delay actually prevented the application from launching.

In general, the performance numbers of the newer languages and frameworks are "reasonable", actually far more reasonable than I anticipated prior to the testing. While we can quibble on detail numbers, the bottom line is (with the exception of the JitC scenario) that the CPU performance is good enough in an era where CPU performance is less important than in the past. Performance is not really a criteria for selecting language/framework at this time. Size of the footprint and in memory consumption is far more important in modern systems. I think that Microsoft has opportunities to

better fine tune the performance tent of universal apps in the future if and when it becomes important to them.

Microsoft's approach in Windows Universal apps forces pre-compilation of the IL, and all of the dependencies, as part of the build process, eliminating the big Jit problem in the future. They are playing with memory options such as compression and app suspension to ease the burden placed on memory, especially for smaller devices; we should expect that they will continue to evolve techniques in the future.

PERFORMANCE IMPACTS NOT ADDRESSED IN THIS TESTING

One of the advantages of managed code over unmanaged code is that it takes the job of managing dynamic memory (typically used for data) out of the hands of the programmer.

Unused memory that is freed up by the programmer in unmanaged code is simply returned. If that memory was used for data, it was tracked by the system in a private memory page and will be freed and (quickly) zeroed.

In managed code, the same data is automatically detected when the last use of the data goes out of scope (explicit dismissal or return from a method that used it locally). But the private memory doesn't actually go away until the Garbage Collection gets triggered.

In this testing, we avoided the allocation and freeing of memory to avoid the impact of GC. All allocation is for potentially sharable code and no freeing of memory is performed during the time of test.

In evaluating which programming language/framework to use for a project, this impact would need to be considered.