PERFORMANCE VERSES DATA

AN ANALYSIS OF THE PERFORMANCE OF DIFFERENT PRIVATE DATA TYPES ON RUNTIME SYSTEMS

Part 1 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 1, attempts to demonstrate some of the new memory handling capabilities included in the Fall 2015 "Threshold" release of Windows 10 and updated in "Redstone". The paper is less about differences in programing languages and frameworks, although some of the testing involves different frameworks to highlight differences that appear in runtime behavior. Specifically, we look at the performance impacts of:

- Memory – In Use
- Memory – When trimmed from Working Set
- Memory – When Page Fault Restored to Working Set.
- Page File
- Time of Execution is certain scenarios

Secondary use and reuse of memory is not addressed in this paper. The datasets used are private process data and would not be sharable. The Part 2 paper looks at code in memory, which is sharable.

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

- .Net C# code in native compiled form
- Windows Universal App C# code

INTRODUCTION

How does the operating system handle data in memory? Is it different for different kinds of data? Or different kinds of applications? Are there environmental triggers that affect the behavior?

These questions came to mind while researching some of the changes made by Microsoft in Windows 10 updates in the Fall of 2015. I had detected some differences in operating system behavior by different apps, and upon investigation determined that the choice of development frameworks were the cause of the differences that I had detected.

Specifically, I noticed an unexpected change in memory utilization in a Windows Universal Application that I was developing when I minimized the application to read my email. I could not account for what happened to the memory! After a bit of research, I stumbled upon a video on Microsoft's Channel 9 that demonstrated something new in the Threshold release they called memory compression.

The implication, which I mostly disprove in this paper, was that it only affected Windows Universal Applications, and I wanted to show how this worked in an updated version of my "Inside the OS" presentations that I give at conferences from time to time.

BACKGROUND ON MEMORY UNDER WINDOWS

Windows uses a traditional virtual memory system where the operating system is responsible for managing competing memory use, allowing both memory sharing possibilities as well as oversubscription of available RAM supported by hardware elements such as the Memory Management Unit.

This paper is too short for a deep dive into how Microsoft implements memory management (See the book *Windows Internals* by Russinovich and others for a deep dive), but here are some basics to consider.

Virtual memory allows for processes to look at memory in their own process space, and this memory is mapped to actual locations. The location may be in active RAM, or on disk (or both). If two processes use the same thing, in some cases the OS will share that memory and the virtual memory for each process will point to the same place in ram. This primary use of this is dll code sharing where the RAM page is marked read-only or copy-on-write.

Both the operating system and all of its processes have something called Working Sets, which is a list of memory pages allocated for/by the process and is in RAM available for immediate use. The Operating System can remove pages from the Working Set (an operation referred to as "trimming") when it needs to free up physical RAM. If the RAM page represents a mapped page of a dll, it already exists on disk and will read back in if the process needs that page again. Otherwise, the data may be written out to disk in a temporary location known as the page file. Memory pages are automatically restored to the process by the OS when the process attempts to access a page not in its current working set through an operation known as a "Page Fault".

In Vista, Microsoft augmented memory management with a set of 8 "Standby Lists". Standby lists contain memory that is not part of any current working set, but might be useful. For the most part, the lists contain pages that were sharable code pages that have been removed from working sets, possibly by the termination of the last application to use the memory page. SuperFetch was also added to proactively read in pages expected to be of use in the future. The memory in standby lists is considered to be relatively "available"[1] in that if the OS runs out of zero and free memory it can be immediately zeroed and given to an application without any disk access required.

What is new in Windows 10 "Threshold" is another form of memory management when pages are trimmed from the working set. Rather than page it out to disk, or remain taking its full space in the standby lists, the OS can compress the pages in memory, and upon the need to restore the page it can perform decompression. This can delay, or even prevent, the need to actually write the page to the page file. But if it becomes necessary to page the memory out, it will do so in the compressed form, reducing the IO overhead. With more than enough CPU, and often plenty of RAM, in our systems, this seems to be a very reasonable approach to me.

[1] Some tools include Standby memory in a category called "Available Memory" and some do not. It varies by the tool as the OS kernel does not have such a singular designation.

Microsoft talks about Memory Compression in relation to modern style applications (typified by Windows Universal Apps), which for the most part remain as processes and in memory when the user clicks away to use another application. The design guides for these apps heavily discourage developers from including an Exit button! Instead, the operating system suspends the foreground threads of the application, and now the entirety of the application working space could be subject to removal (although currently it is only the private memory that is targeted automatically). Keeping all of that memory in RAM for quicker restoral when the user returns to the application is the motivation for compression.

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 created a new set of bad application tools, and updated an existing monitor that I had written to investigate performance impacts of new memory features in Windows Vista a dozen years ago.

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 two era of languages/platforms:

- A "Microsoft.Net" style application written in C#. The Microsoft.Net 4.5.2 Framework was used, and the exe was built as "x64". NGEN compilation was performed on the app to be more consistent with the other app.
- A "Windows Universal" application written in C#. The app uses a Universal App front end and the Windows RT framework, and the exe was built as x64.

A non-managed C++ form of the code was not created or tested. Prior work on older operating systems indicate that such a program should be treated pretty much the same as the .Net C# code. Proof of this is left to the reader.

TEST APPLICATIONS

Two programs, PrivateMemoryEaterWPF and PrivateMemoryEater_UA, were created to be as identical as is possible.

PrivateMemoryEaterWPF is written as a .Net 4.5.2 C# application. PrivateMemoryEater_UA is written as a Windows Universal App also in C#. Although some prior testing had hinted at the possibility that these would be treated differently by the OS, this more careful testing showed this not to be the case.

Both applications have a dialog interface that allows the user to designate an amount of memory for the application to allocate and fill with pattern data. As I like to work big the input is in megabytes and the apps were built as 64-bit so that allocation of multi-gigabytes were possible.

Due to a new feature in Windows 10 (Threshold release) called memory compression, thee forms of data filling is supported:

- **Uncompressible (Default).** A random value is written into each byte.
- **Semi-Compressible.** The same value is written in blocks of 20 bytes, then a new random value chosen for the next 20.
- **Compressible.** The same value is written into all bytes.

Two additional buttons are provided to aid in the testing:

- A "read data" button that will cause each byte of the allocated memory to be read. This will force the memory back into the working set of the process on demand.
- A "deallocate" button that will release the memory and call upon the garbage collection to immediately return that memory to the system.

THE MONITORING TOOLS

For monitoring, my ATM[2] tool provides a convenient graphical depiction of the memory use over time at a detailed level, and ultimately is useful to provide context regarding how the numbers changed over time.

ATM was updated (version 4.6) for this work to support the selection of a single process to be selected so that separation of its memory use from the general used memory may be seen. Also a button was added that can be used to request that the memory used by that process be removed from its working set. This button simulates what the OS might do if available memory became too low. ATM also has a feature allowing for on-demand flushing of all of the memory pages from the working space of a target application.

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

[2] Free tool available from www.tmurgent.com

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 WPF APPLICATION

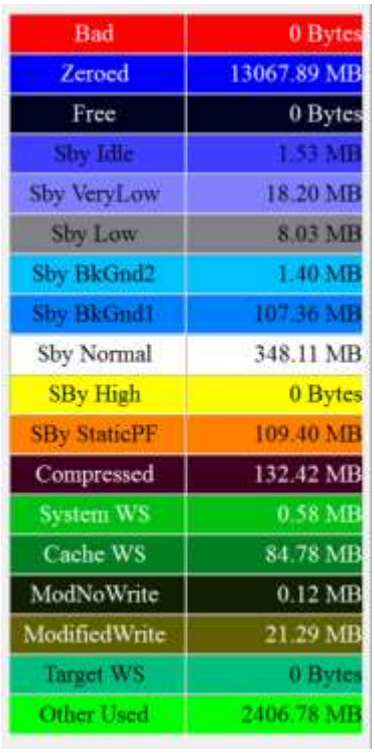
"TYPICAL" COMPRESSABLE DATASET IN WPF

In this test case, 5GB of memory was allocated by the test application and filled with data that should provide a fairly good compression. Microsoft appears to be performing the compression on each 4k memory page separately, which probably reduces the expectable compression ratios somewhat over the more typical 64k block compression techniques. It is assumed that they use the ZLib compression that is built into the kernel and used for other OS activities.

The images used in this paper are taken from the ATM tool. Generally, the images are a timelines and will have a distinct upper and lower portions.

The top portion shows RAM memory usage via a number of categories as shown in the key on the right. The "teal" Target WS shows the working space of our target application over time. The "maroon" Compressed category shows the working space of the new Memory Compression process. The brownish green shows the size of the Modified Write queue. The white band is the size of the "Normal" Standby List (not really involved in this data example with WPF, but is involved with Windows Universal App data and in memory mapped code pages for all apps).

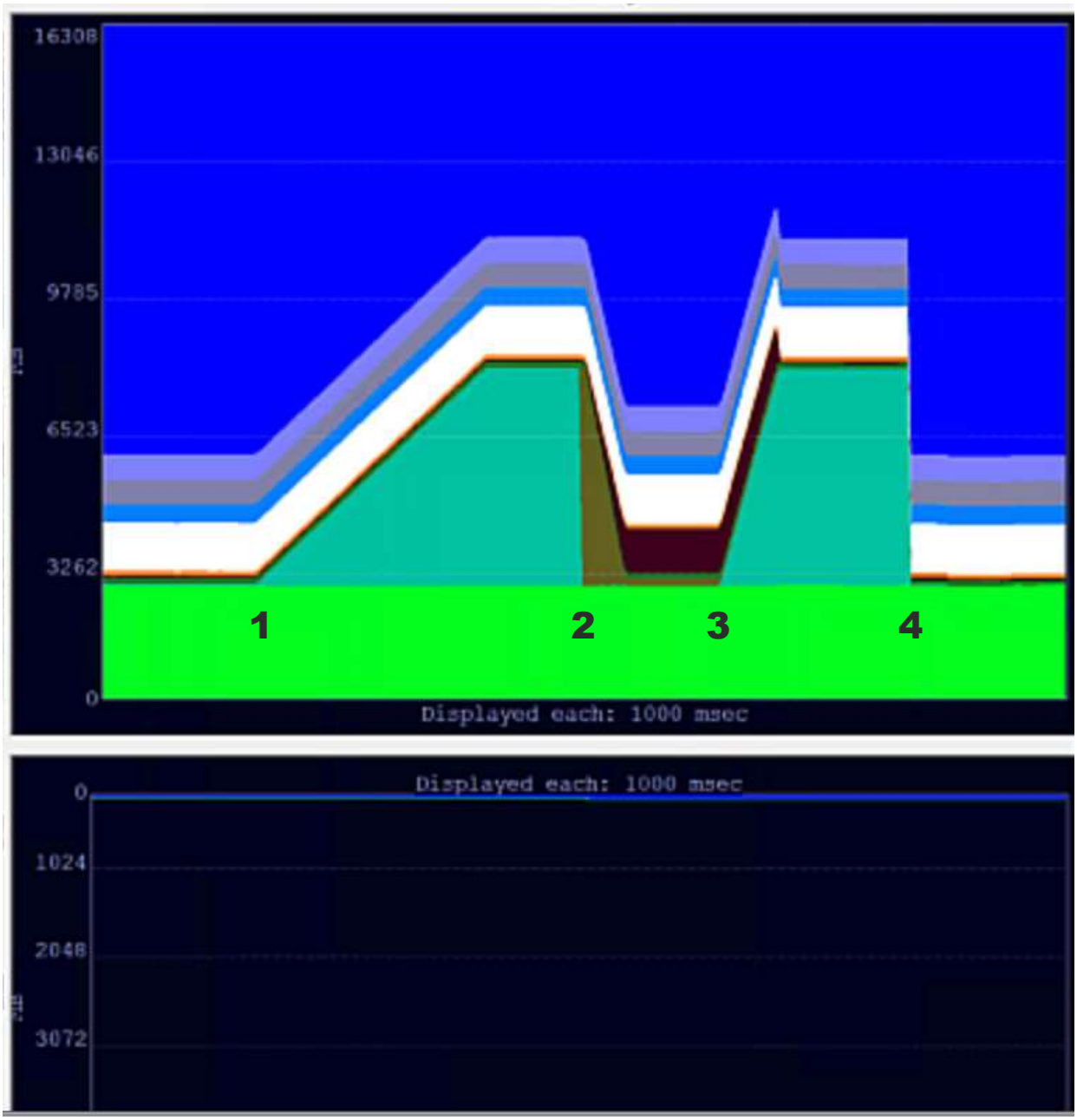
Event numbers are often overlayed on the upper portion that will relate to the event descriptions that accompany the image.



The bottom image portion, when present, shows page file usage along the same timeline. Because the system has plenty of available RAM, in the WPF tests the page file is also not really used. We will see this come into play with the Windows Universal Apps later.

The image on the following page shows the results of allocating and filling the data, what happens when the process has the memory trimmed from its working set, restored to the working set, and released.

Figure 1- WPF with Semi-Compressible Data



The indicated events are as follows:

Event 1: Target Application requests and fills the 5GB of data. (Four 1GB requests are made). We can see the memory usage of the target process grow by the 5GB. As there is plenty of free memory on the system, other memory usage remains relatively stable.

Event 2: Removal of memory from the target process only. Although the artificial method of using ATM to request the removal is used, subsequent testing of processes in scenarios with available memory constraints prove that the results of this artificial request are consistent with real-world experiences.

We can see that immediately upon event 2, all of the private memory of the target application is placed into the modified write queue. We also see that data move from the modified write queue into the working space of the new Memory Compression process, however at a much smaller size, about 1/5th of the size of the original data. During this time, the Memory Compression processes is consuming one of the available CPUs, so obviously it is responsible for performing the compression. Unlike event 3 which follows, no significant jump in overall memory use occurs, hinting to each page getting compressed and the old page freed up immediately. Nothing is written to the page file in this example, so obviously compression must be in play. This is confirmed by the additional tests that follow using different data sets.

Event 3: Target application attempts to read access every byte of the 5GB. This forces the pages back into the working space, decompressing along the way. It appears that the Memory Compression process is involved, however it only consumes about 10% of the CPU used to compress. It seems likely that decompression would take less CPU, but I am guessing that the CPU to decompress is charged to the target process as part of the page fault[3]. Notice the memory spike in the decompression case. It appears that the memory is not immediately released. Quite possibly, the decompression into new memory occurs as part of the target process page fault, and the Memory Compression process must wake up to release the compressed form.

Event 4: The target application releases the memory. As this was private data memory, it would be placed in the free pool. From there, the threads of the Idle System process (ProcessID 0) would zero out the memory and place into the zero memory pool. Obviously this happens very fast as we cannot detect the free pool increasing. The system idle process has as many threads as logical processors in the OS (4 in our case), and they can work in parallel. The system was relatively idle so at least 3 of them would have been operating full time.

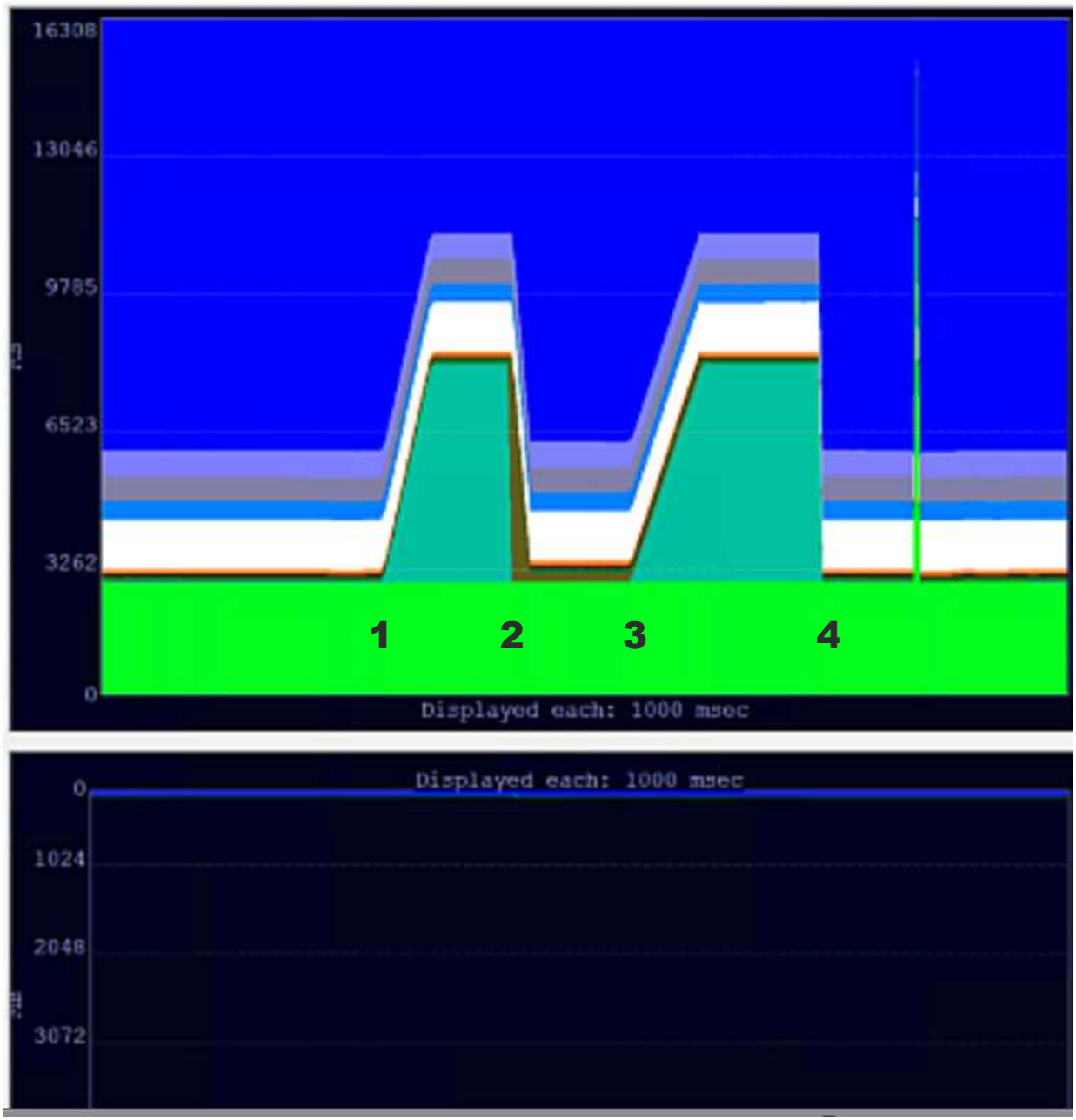
"VERY" COMPRESSABLE DATASET IN WPF

In this test case, 5GB of memory was allocated by the test application and filled with identical data that should provide a very good compression.

The image on the following page shows the results.

[3] Because the target application is reading and adding up a hash of the data during this period, it is consuming a full CPU also, so it is difficult to tell.

Figure 2 - WPF Compressible



Event 1: This dataset takes less time to allocate and fill, but only because the target application is not generating a new random numbers.

Event 2: Event 2 is similar, except that the compression ratio is so much greater. The 5GB compressed down to a small amount of memory, about 23MB, and in less time (about ½).

Event 3: Event 3 takes almost as much time as previous to restore the memory. The memory spike we saw earlier is not noticeable here, but that is because the "extra" uncompressed copy is so small in proportion.

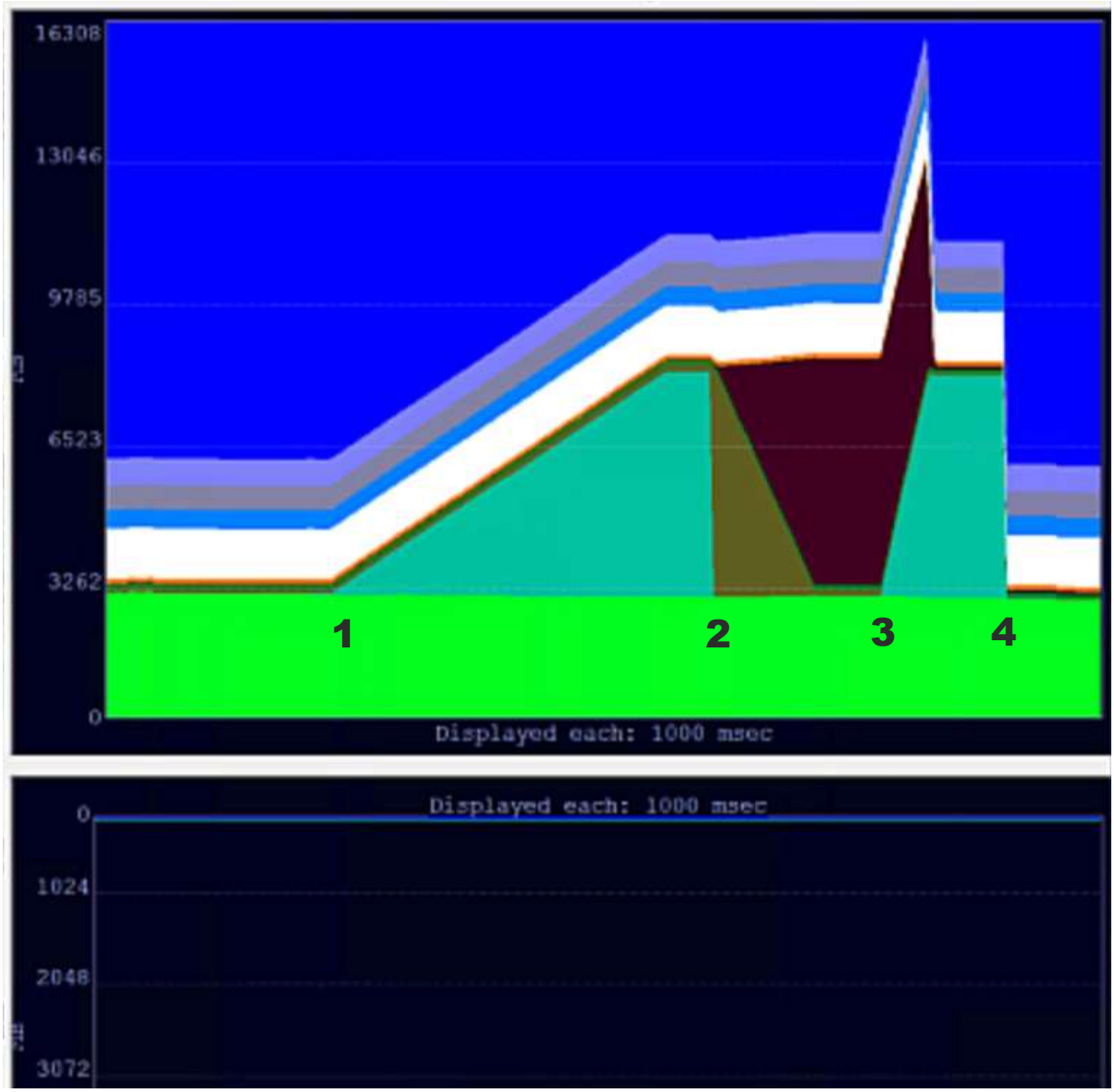
Event 4: Event 4 is boringly similar to the prior example.

"UN" COMPRESSABLE DATASET IN WPF

In this test case, 5GB of memory was allocated by the test application and filled with completely random data. Typically, this type of data will result in a compression size greater than the uncompressed data. Often, a decent compression algorithm will notice this and slap on a compression header indicating that the data actually isn't compressed, resulting in the compression being only slightly larger than the original. Encrypted data often acts the same way as random data when it comes to compression.

The image on the following page shows the results.

Figure 3 - WPF Un-Compressible



Event 1: Event 1 takes longer only because the target application is generating a new random number for each byte.

Event 2: Event 2 takes longer to compress the random data, almost twice as long as for the semi-compressible data. Also notice that the memory used is slightly larger than prior to compression, as predicted.

Event 3: Event 3 took only slightly less time than for the semi-compressed data, rather than more. This would seem to indicate that perhaps the compression ended up with generally uncompressed data. The overall memory spike during decompression is quite pronounced in this example.

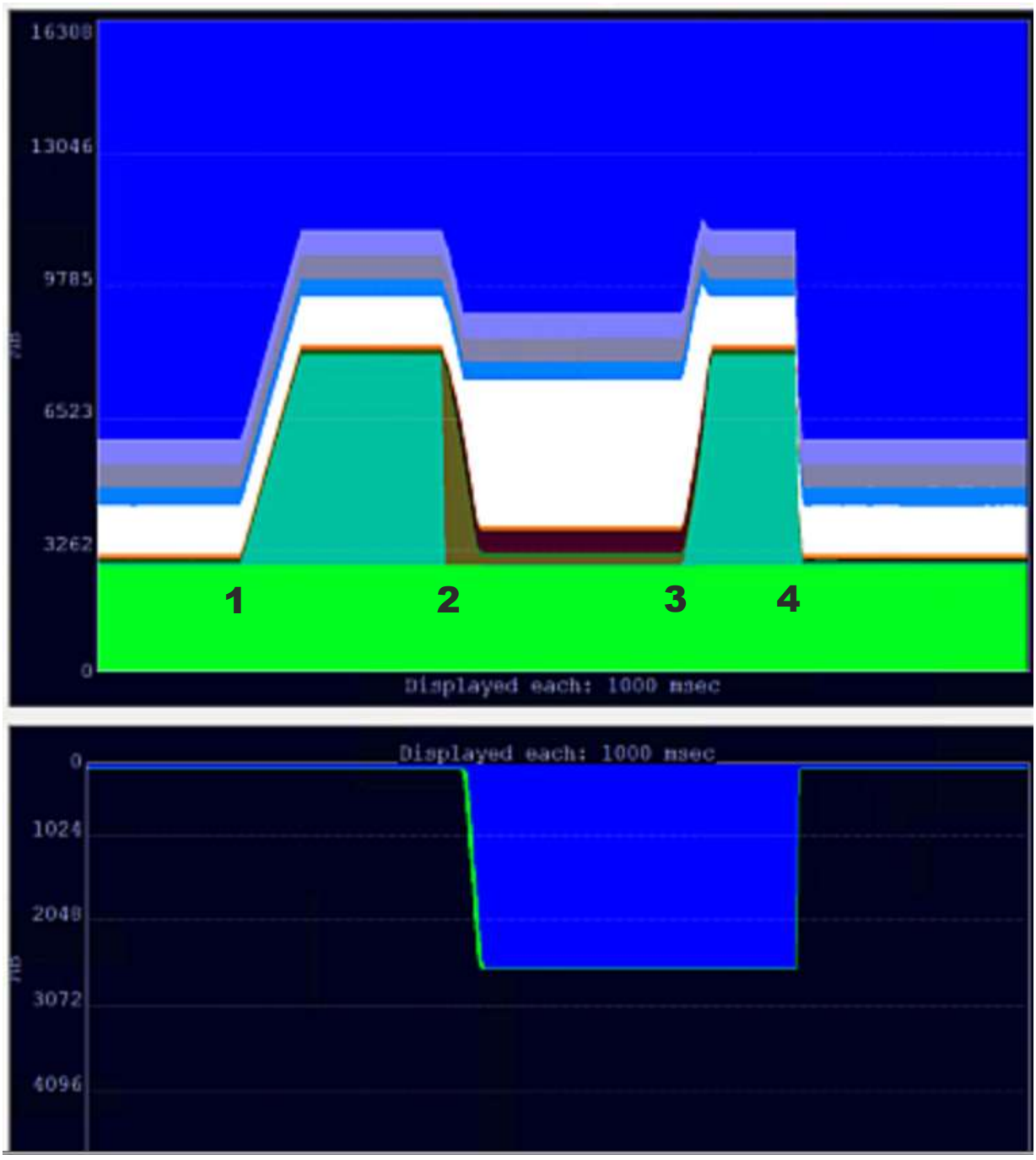
Event 4: Event 4 looks normal.

TEST RESULTS FOR WUA APPLICATION

"TYPICAL" COMPRESSABLE DATASET IN WUA

In this test case, 5GB of memory was allocated by the test application and filled with data that should provide a fairly good compression. The image on the following page shows the results.

Figure 4 - WUA Semi-Compressible



We can quickly see that, even in the absence of a memory need, the Windows Universal App is treated differently than WPF (and unmanaged "Win32") apps.

Event 1: Event 1 occurs much like it does for the WPF counterpart. As is shown in Part 2 of this white paper series, there may be a slight speedup of the target application code execution, but it is too small to be reliably measured in this test as the test is more dominated by OS activity.

Event 2: Event 2 is where differences occur[4]. As before, the memory is removed from the working space of the target application and can be seen in the Modified Write queue for a short period of time. The Memory Compression processes takes the memory and compresses it. But with WUA apps, it seems to leave some of the compressed memory in the working set of the Memory Compression process, and page the rest out to the page file. In addition, we see the Normal Standby list grow.

In the WPF case for semi-compressible, the 5GB was compressed to just about about 1GB and remained in Compressible Memory working space[5]. In the WUA case, compressible memory increase was about 0.5GB. Assuming that this represented about half of the original 5G, then increases in the page file and standby list of 2.5GB imply that the Compressible Memory algorithms detected that this came from a WUA app and capped the compression to 0.5GB, then paged the remainder out to the page file. Then it placed the uncompressed memory in the standby list.

Event 3: Event 3. Because half of the memory to be restored to the workspace was still present (in uncompressed form) in the Standby List, this memory could be restored via traditional "soft" page fault – that is no memory was moved, just the tracking of it. The page file is not modified as part of the restoration; this is normal page file behavior for restored data.

Event 4: Event 4. The event is just like the WPF counterpart, except when the memory is freed, the entries in the page file are no longer required and are removed.

"VERY" COMPRESSABLE DATASET IN WUA

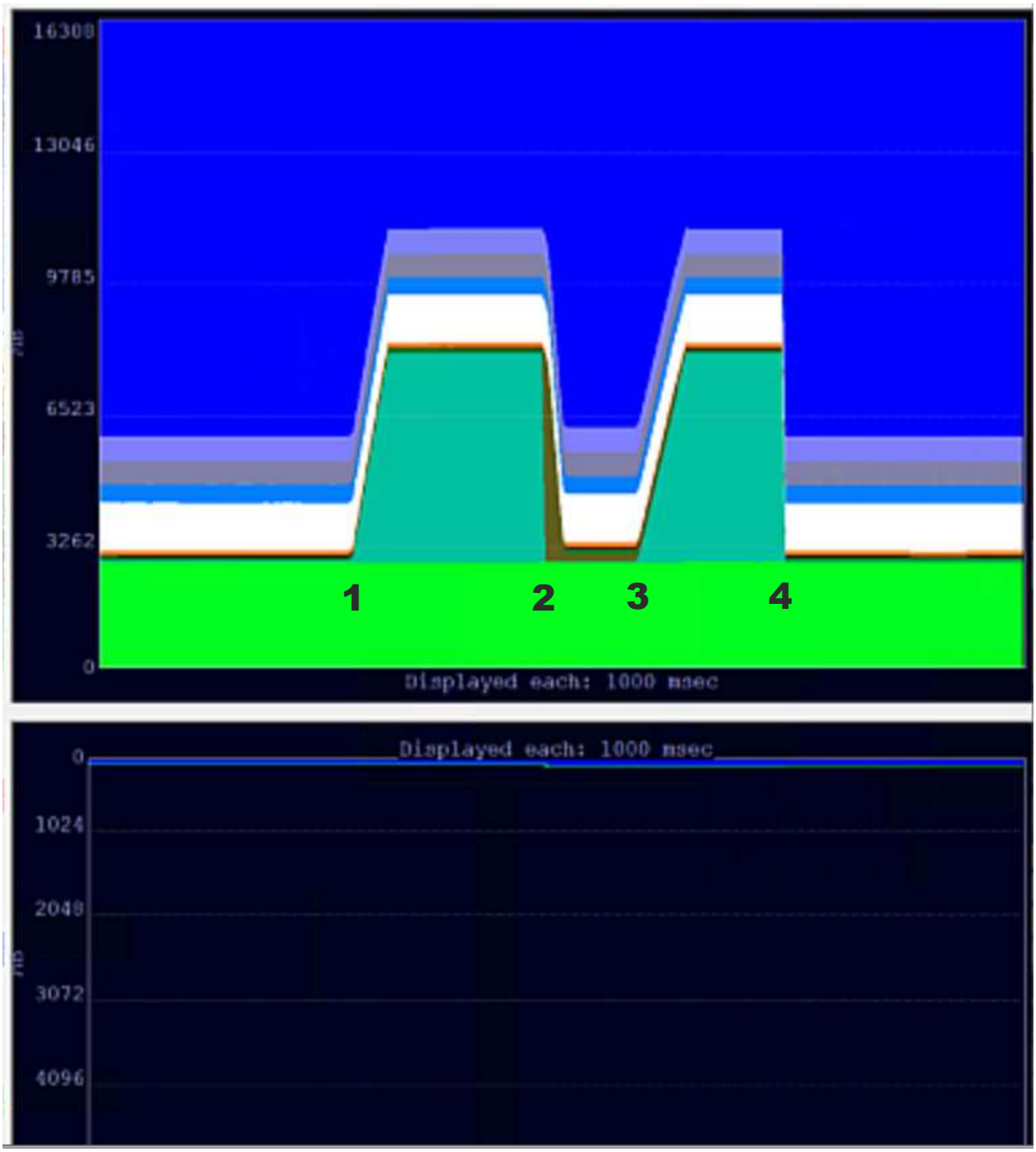
In this test case, 5GB of memory was allocated by the test application and filled with identical data that should provide a very good compression.

The image on the following page shows the results.

[4] I began this testing on Windows 10 "Threshold" release, which behaved quite differently than for the "Redstone" release which is used in this paper. This is especially seen at event 2, which would previously have behaved more like the WPF example, except it would have done so as soon as the target process was suspended, which it no longer does. Microsoft is evolving their memory management strategy, and future versions of the OS may behave differently.

[5] A small increase of about 1MB was observed in the WPF test to the page file, but this was likely not our data.

Figure 5 - WUA Compressible

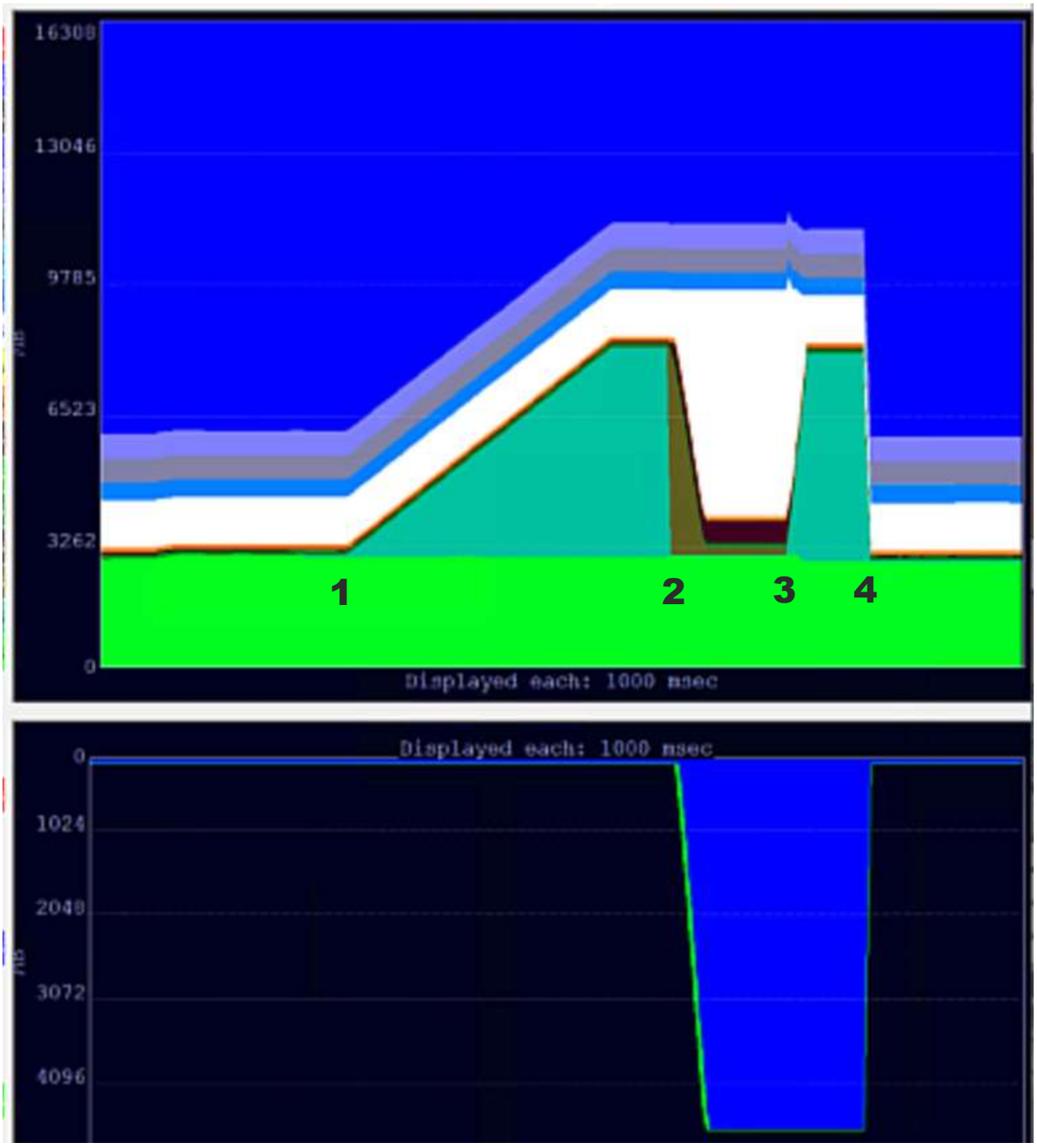


Event 2: During Event 2, the Memory Compression process working space grew by only 19MB. The assumption here is that because the compressed size was so small, it just kept it in the compressed memory process working space. The page file also grew about 20MB, but this is probably code pages of the process being paged out.

"UN" COMPRESSABLE DATASET IN WUA

In this test case, 5GB of memory was allocated by the test application and filled with completely random data. The image that follows shows the test results.

Figure 6 - WUA Un-Compressible

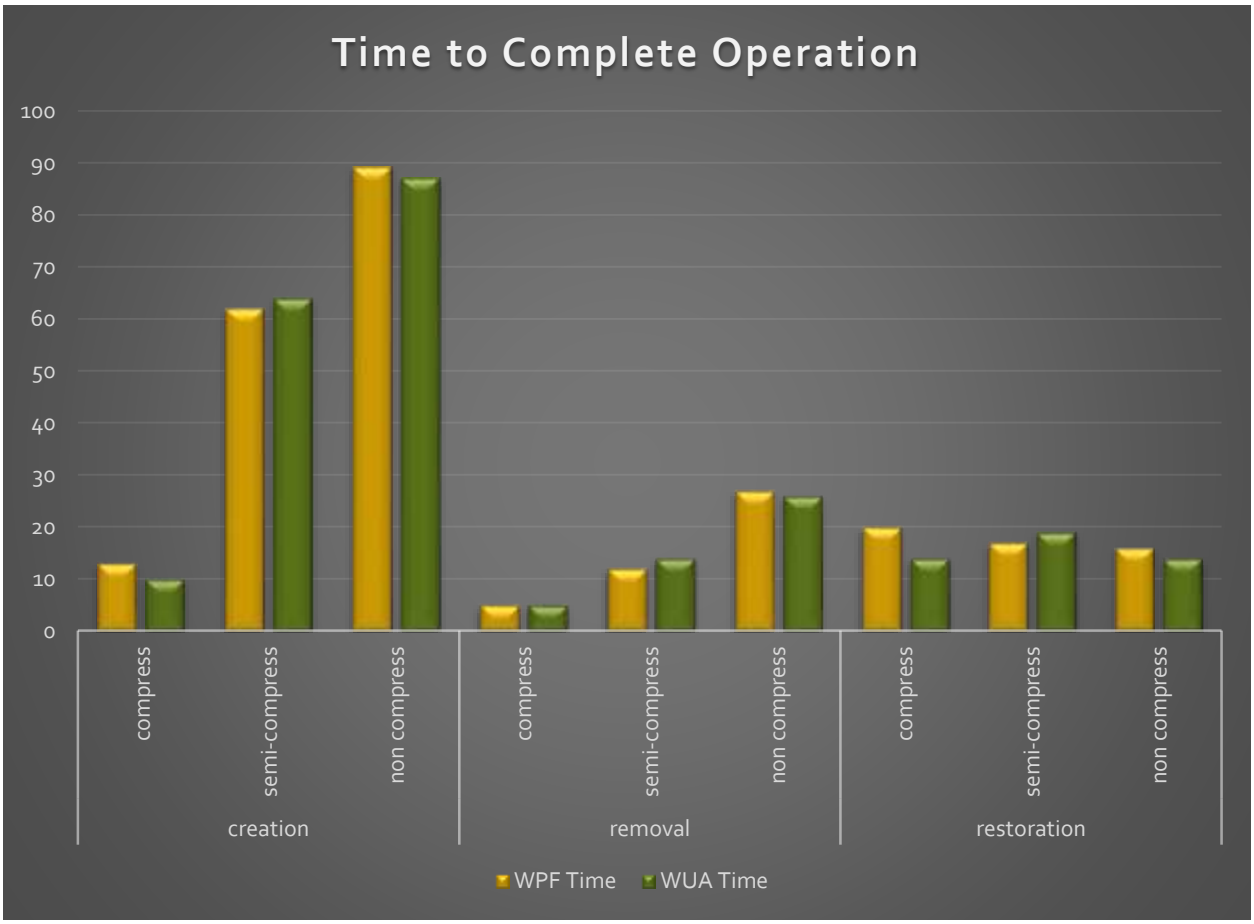


Event 2: When compression occurs at Event 2, we again see the compressed memory process cap out at about ½ GB increase. The page file and standby normal list both grew by 4445MB, and presumably this represents the uncompressed remainder of the original working space. The restoration time for the WUA app was really fast (about 3x faster than for the WPF app where all the data had to be uncompressed).

ANALYSIS OF TEST RESULTS IN THE ABSENSE OF MEMORY CONTENTION

The purpose of memory compression is to reduce the likelihood of paging operations during hard page faults. The tests shown in this paper do not show situations where this comes into play; this was done so as to make the actions being taken by the system easier to understand. In the absence of a compelling need to free up RAM, any form of compression would result in added overhead. Please don't take these results as condemning memory compression; we just aren't testing it on memory constrained systems where it is needed.

The chart below shows the time taken to perform certain actions (events 1, 2, and 3 of the prior charts).

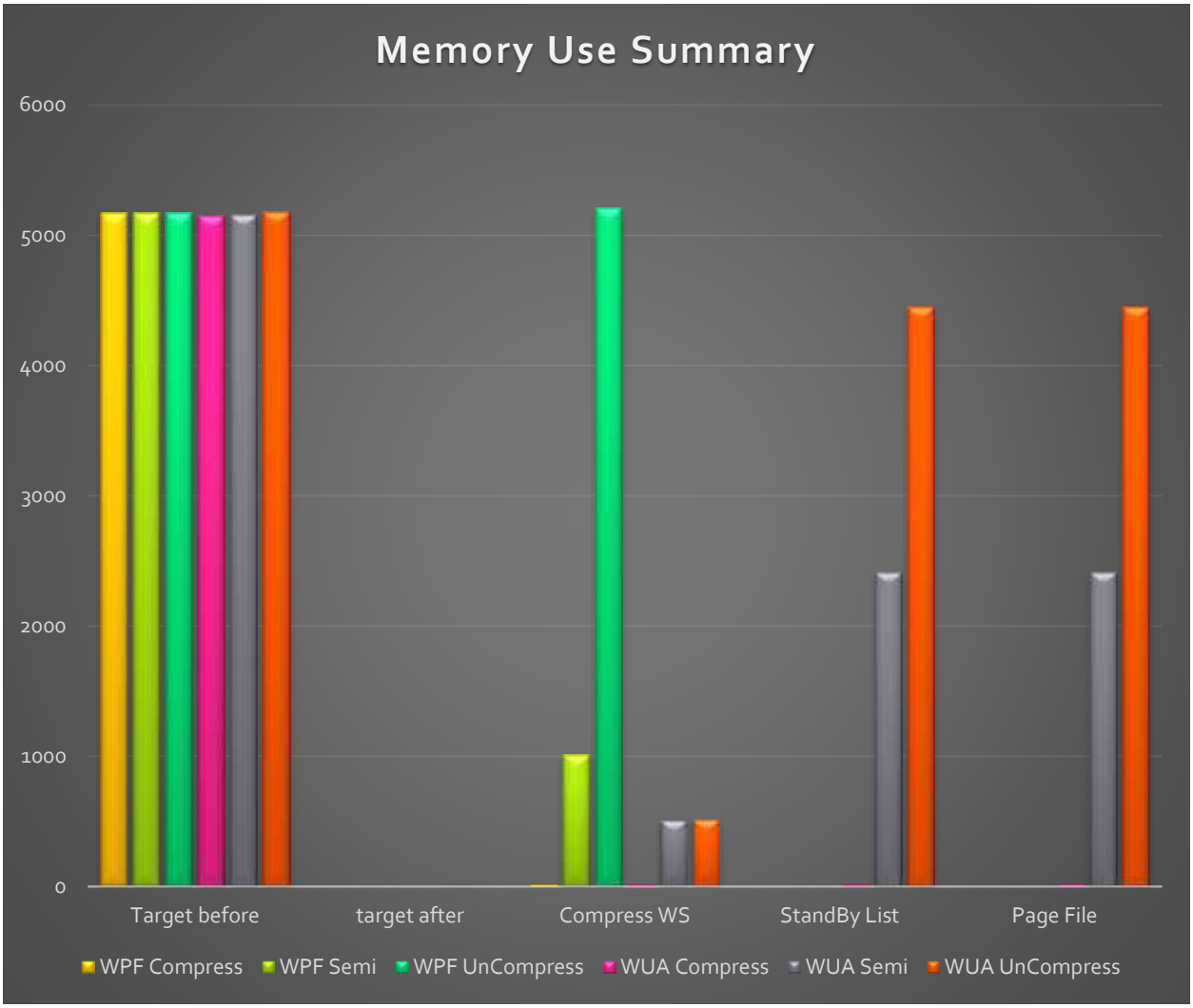


The differences in creation time on the different data sets is accounted for by the target application needing to generate more random numbers to create less compressible data.

It also appears that the time required to compress increases as the data is more random. This is probably due to the larger dictionary lists to search through when the data is more random.

Restoral times should be faster in situations where more data is left in the standby list in uncompressed form, which occurs only for the WUA semi and non compressible cases. The comparison of test results for the restoral situation don't show this, hinting that there may be more in play during restoral than we understand at present.

The chart below summarizes the memory/page file use situations.

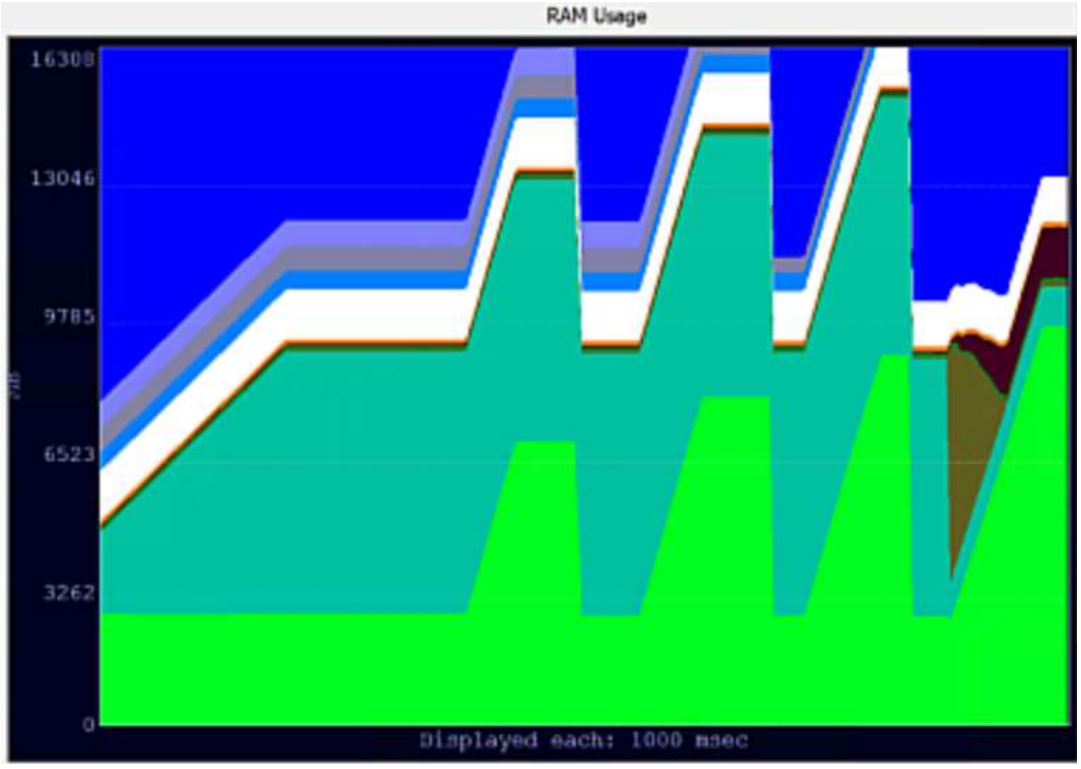


That compression is limited to the first 512MB (after compression) for WUA is probably not that important. The reality of WUA apps is that they are very unlikely to have that much data at this time, so the expected effect on most WUA apps would be the same as for WPF.

ADDITIONAL TESTS WITH MEMORY CONTENTION

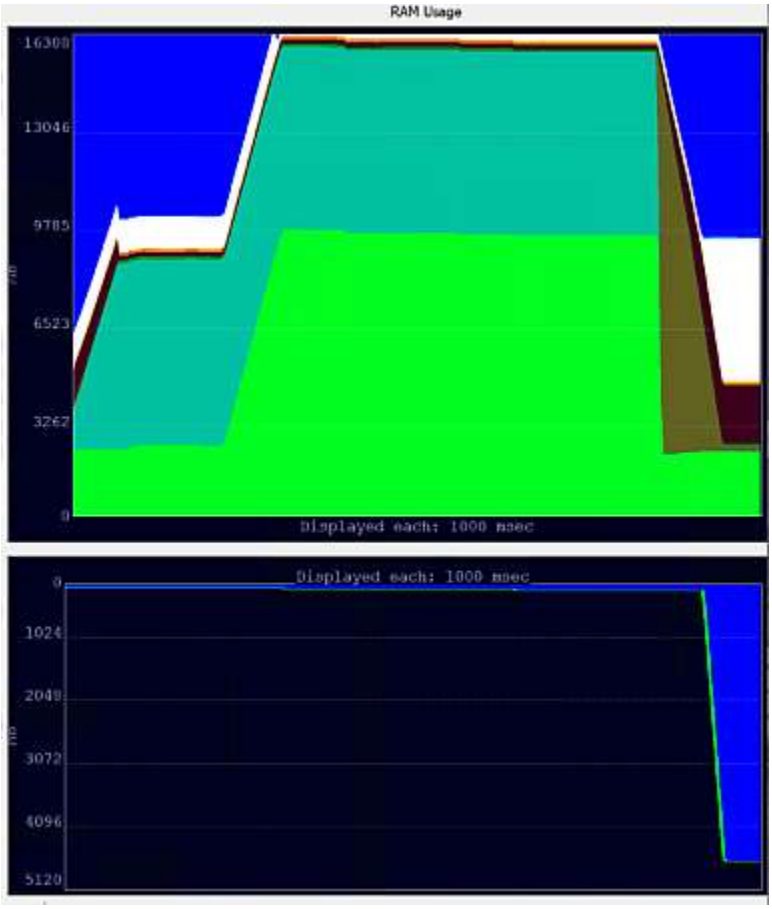
Additional testing was performed on Redstone with both apps in situations where memory could become more constrained. The details of these tests are not presented here, however, we can summarize that when an app is requesting memory and the available memory is getting low:

- The system will free up memory from the standby lists, starting with the lowest priority memory,
- The system may trigger trimming of pages from the working space of some applications. The algorithm for choosing which app to target is not obvious.
- When the system decides to trim, it seems to target the entire application, much like the ATM test tool. This is different behavior than older versions of Windows and I don't know when this changed.
- If a WPF app is in memory and not active (whether or not minimized), the trigger to trim working sets does not seem to occur until free memory is to become exhausted. In the test shows below, the target application is our WPF app, and then the WUA app requests memory of increasing size until we see something happen. When the app requests the memory, the virtual memory is immediately reserved and then added to the working space of the app as it starts writing into the memory[6]. In the image below, we can see that it is when the secondary app request 7GB of memory, which would exceed the zero+free+standby memory, that the trimming occurs.



[6] This represents the major difference from an unmanaged Win32 app where all memory would be added to the working space immediately upon request.

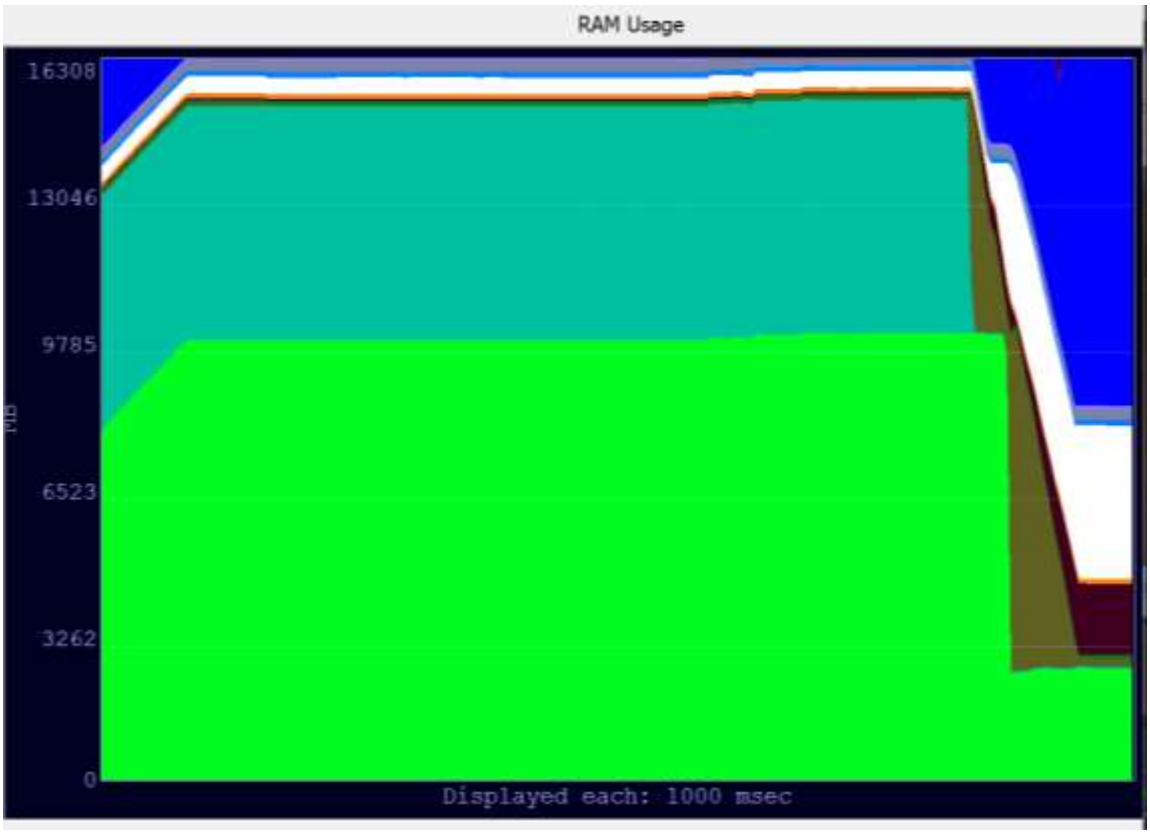
If the new application request something near the limit, no immediate trigger occurs, however, other OS activity might cause it in the future. This is seen in the image below:



In this example, both apps are active. After the allocation and filling, the OS does seem to be trimming some memory from the OS or other applications, and the normal standby list is slowly increasing in size.

Some unknown activity seemed to trigger trimming of BOTH apps, with the WPF app pages ending up in Compressed Memory and the WUA app with 1/2GB in Compressed memory and the remainder in the page file and Standby lists.

- If a WUA app is in memory and not active and memory is not limited, nothing special occurs when it is suspended in the Redstone release (in threshold, it would have been immediately trimmed). If memory is in short supply, suspension will cause trimming. This is seen in the following image where the WUA app is the target app and is suspended to trigger the trimming of its working space. Surprisingly, this subsequently triggered the WPF app to be triggered shortly after (but not at the same time).


RAM Usage
16308
13046
9785
6523
3262
0
MB
Displayed each: 1000 msec

INTERPRETATION

Microsoft seems to be trying different things to improve memory performance using compression. It would not be surprising to see additional tweaks in the future.

In particular, Microsoft is trying to treat WUA apps differently than other apps. This would seem mostly due to the ways in which users interact with those apps on phones and tablets where they never close the app but just suspend it to work on something else. And they are targeting restoral time as being the most import factor.

In working with WUA apps on a Windows 10 device in desktop mode, however, I do not find myself working the same way. WUA apps and traditional apps look pretty much the same to me, except for the missing close button in the GUI – but I just use the X button on the titlebar.

In the desktop mode, we may see Microsoft remove the app type distinction, although in which way is anybody's guess.