A look at Windows WPF Application Performance with MSIX

Research Tests to understand typical scenarios and underlying causes

Tim Mangan

TMurgent Technologies, LLP

August 2026

Executive Summary

This research paper looks at the performance of WPF .Net Framework applications when running in an MSIX container, with and without the Package Support Framework (PSF). Both launch performance and runtime performance characteristics are considered.

Testing was performed on purpose-built application loads to provide comparable data under a set of different circumstances. Real-world application testing was not part of this research to better understand the causes of any performance issues, but from the tests run we make predictions on most real-world apps. A second paper on WinUI applications is also planned. The primary findings include:

- The overall performance of MSIX based applications are not much different than the native app counterpart. Only file creation (not writing) was implicated in performance difference. This was not the case when we tested App-V in the past[1], which had performance differences on certain registry operations.
- Time to launch an application in the that has no significant app-specific launch processing in the MSIX container is slightly longer than that of the native launch, but well below the level detectable by an end-user.
- Launching with the Package Support Framework PsfLauncher surprisingly added no additional time to launch the application (with the margin of error), even though we have two serialized app launches being measured when in the container.
- Adding the PSF PsfLauncher and all PSF fixups also showed no user-noticeable difference. This result was a surprise as the fixups are known to in some cases be optimized to use more time at initialization to save runtime overhead later on. Prior to running these tests I had assumed there might be a need to alter that strategy, but the results say otherwise.
- The runtime performance of running in the MSIX container for registry operations show a very small degradation when running in the container, with or without the PSF. In these tests the difference does not appear to be noticeable by the end-user.
- The runtime performance of running in the MSIX container for file operations did show a small degradation over native performance for enumeration/read, but I doubt that it would noticeable by the end-user in production apps.
- The runtime performance of running in the MSIX container for file write operations (file creation, not actually writing file contents) does show a performance impact. The impact of InstalledLocationVirtualization was 5x, and the MFRFixup was

[1] See https://tmurgent.com/AppV/en/resources/research-and-white-papers/91-app-v-5-research/293-deploymentperformance

between 6 and 7x. Keep in mind that this is only applicable to file creation and opening for write, the performance of writing data into the file should not be affected. Applications generally do not open that many files for write purposes, but of everything tested so far, the file creation of the ILV and MfrFixup software look like the best targets to target for performance improvements for the future.

Introduction

For more than eight years, I have worked on making traditional applications run properly inside the MSIX container. What was once a significant challenge is now much easier, and we are seeing more people working with this technology.

With that success, this summer we have seen some folks turn to investigating the performance aspects of MSIX; producing some tests that attempt to show how much overhead there is. We should, of course, expect some level of overhead in putting applications into the isolation container. After all, you can't obtain the benefits of running in a container without expecting a bit of overhead.

For a parallel example, placing the operating system into a virtual machine introduces overhead. In a well-designed, lightly loaded environment with a single VM allocated with equivalent hardware resources to the native machine, modern hardware-assisted virtualization is often close to native performance, but it is not free: published benchmark summaries commonly place optimized VM overhead at roughly 5–10% overall, with low single-digit CPU overhead for many compute-bound workloads and larger penalties when storage, network I/O, contention, or non-optimized configurations dominate. Microsoft's Hyper-V tuning guidance similarly notes that virtualization servers share and virtualize processors, memory, and I/O resources, and that Hyper-V hosts can require increased CPU, memory, and I/O capacity compared with non-virtualized systems. (See references [2] [3] [4]) Under Microsoft App-V, I previously took a look at the performance there with a series of research papers (See [5]).

When trying to quantify the performance of an application, we need to look at two major performance scenarios, overhead associated with launching the application and overhead associated with the running application. To minimize the impact of measurement, the testing done for this paper utilizes a full launch and shutdown of the app without added workload as a fair estimation of launch time as the time to shutdown the application does not vary that much. To determine the runtime impact of a given workload, the launch and

---

[2] **Performance losses with virtualization: Comparing bare metal to VMs and containers** https://jojoc4.ch/publications/tm.pdf

[3] **Performance Tuning Hyper-V Servers** https://learn.microsoft.com/en-us/windows-server/administration/performance-tuning/role/hyper-v-server/

[4] **Hyper-V Configuration** https://learn.microsoft.com/en-us/windows-server/administration/performance-tuning/role/hyper-v-server/configuration

[5] **App-V 5 Research** https://tmurgent.com/appv/en/resources/research-and-white-papers/91-app-v-5-research

shutdown time of the application executing without a workload is subtracted from the time of executing with a workload added.

In App-V, using this technique showed significant overhead in launching the application, due primarily to the need to set up the App-V container but also affected by whether the app was fully cached locally before the test. When we eliminate the latter by fully caching, we would typically see an additional 1 to 2 seconds for the application window of a small test app to appear. The actual amount depended quite a bit on what the app itself did in the startup initialization, leading to a few production apps having significant delay in the window being ready. Once the app was running, tests showed only a minor performance difference in most apps. The impact there would be due to having to look for registry and file items in multiple locations (up to 3) whereas the native call would only look in one place. But the difference depended on the app and ranged from an unmeasurable difference to up to 5% performance degradation, depending on the kind of workload involved.

To look at the Performance impact of MSIX, four scenarios are considered:

Test Scenarios
• **The application runs natively without a container.**
• **The application runs in an MSIX container without the Package Support Framework (PSF). InstalledLocationVirtualization is enabled so that the file write tests can succeed.**
• **The application runs in an MSIX container with the PSF launcher only. InstalledLocationVirtualization is enabled so that the file write tests can succeed.**
• **The application runs in an MSIX container with all PSF components but without InstalledLocationVirtualization.**

In such performance testing, care must be taken to eliminate interference in the tests. When I perform such testing, this includes:

- A well-prepared environment prevents as much external noise in the environment that can affect the machine, such as external management agents.

- A physical machine is ideal, but a VM is acceptable if its virtual resources match those of a typical user's device and no other VMs are running on the same hypervisor.
- The test machine/VM should have services and background tasks minimized as is best practical.
- Testing should be repeated multiple times with the ability to remove outliers from the results in case something happens in the background affecting a particular run. Similarly, it is normal to eliminate the first test run as windows caching (see my book "Windows System Performance Though Caching [6]).
- Tests should be run after the system is booted, and the user logged in, followed by a sufficient amount of time passed for the system to settle before any tests are run. This allows for automatic background activities that occur after the first user logs in to run.
- Some settling time should also occur between test runs.
- Wherever possible, the tests must be fully automated to eliminate human interference with the results.
- Tests for the 50k file creation required excluding the process name in Windows Defender configuration as MsMpEng would consume all CPU resources. Oddly, this was only needed for the native tests for some reason, but I turned it off on all scenarios[7]. Possibly this is because the exe was unsigned and not under Program Files while the package is signed.

Using these principles, the testing on applications under MSIX and the PSF were undertaken for this paper. The source code and scripts for the tests are publicly available in GitHub (See [8]), as is the PSF itself (See [9]).

Rather than perform tests on a set of known applications, the testing here uses a single known test application, written in C# for .Net Framework, with varying payload functions used to act as different applications. The purpose of this is that we are not seeking to look at overall application averages, but to isolate and illuminate specific system and application behaviors. This, in turn, will highlight where specific performance improvements should be targeted in the future to improve MSIX and/or PSF performance.

---

[6] https://tmurgent.com/appv/en/resources/books/80-books/73-windows-system-performance-through-caching

[7] The test app exe was not signed, so perhaps this is why it is better trusted when running inside the signed container.

[8] **TimMangan/TMurgent_LotsOfStuff:** https://github.com/TimMangan/TMurgent_LotsOfStuff

[9] **GitHub - TimMangan/MSIX-PackageSupportFramework**: https://github.com/TimMangan/MSIX-PackageSupportFramework

Although not the intent, the results of this testing shown in this paper may also help others concerned about the performance of a specific application make changes to the application itself to improve performance.

> As an example of how this may help, when we originally investigated App-V performance we located a very high-performance penalty existed for an app that tried to enumerate a large number of registry keys that did not exist. Tracing some applications that had significant startup penalties under App-V showed that this was happening to this app. Microsoft (to our knowledge) never addressed the issue in the App-V client, but the vendor could have significantly sped up their software by checking for existence of the key first (a significant improvement under App-V but also a small improvement natively).

The Test Environment

The test environment used involves a Windows VM on a virtual machine server in a relatively isolated environment.

The hardware:

Intel Ultra 7 155H – 16 cores 22 logical processors

32 GB Ram

2 SSD, one for OS and one for VHDs.

The Hypervisor:

Microsoft Hyper-V (running in Windows 11 Desktop OS)

The VM:

4 vCPU, 8 GB Ram, dynamic range to 10GB

Microsoft Windows 11 Pro 25h2 with Insider Preview. (26300.7965)

VM disk is located on the hypervisor.

Windows Updates disabled.

Many other services and scheduled tasks disabled.

Anti Virus configured to ignore the application under test.

The Application:

In this paper we use a single testbed application. Repeating this kins of test with applications built using different developer frameworks is left to the reader.

LotsOfStuffWPF_DotNetFramework Microsoft .Net Framework 4.8.1 application written in C#.

The app consists of a single window GUI that presents a progress bar and closes when the work requested is done. The app contains a number of different workloads that may be used during the run. A command line argument picks which workload to include in the test.

Because an application's initialization activities tend to be the same kind of activities that runtime workloads do, it is important to have a well known application with a minimal initialization.

The Workloads:

Currently the application has a base workload, and 6 different workloads to be tested in each of the test scenarios.

Test Workloads
Base workload. No additional runtime code**.**
Timer workload. The software will sleep for a total of 5 seconds. This sleep is done as a single sleep of 5000ms. This workload should cause almost no additional activity of the base workload, but will add to the total time run to help validate the workload testing.
Reg Enum 50k. This workload consists of enumerating items from the Windows Registry. Values are not read, but keys and value names are enumerated. The workload will continue until a total of 50,000 items have been enumerated. The items to be enumerated will come from the base OS registry, not the package. Registry calls are in HKCU and avoid permission requests not allowed in MSIX without remediation.
Reg Write 50k. The workload will create a key, then create a total of 50,000 items, including both keys and registry strings. The number of items per key is limited to reasonable size. Registry calls are in HKCU and avoid permission requests not allowed in MSIX without remediation.
Reg Write/Enum 50k. This workload creates the 50, 000 keys/items, and after that enumerates them. From this you can extract the time to enumerate 50k registry enumerations of virtual registry items in the package. Registry calls are in HKCU and avoid permission requests not allowed in MSIX without remediation.
File Enum 50k. This workload consists of enumerating items from the c: drive. Files are not read, but folders and files names are enumerated. The workload will continue until a total of 50,000 items have been enumerated. These files come from the native OS.
File Write 50k. This workload will first create a folder, and then create a total of 50,000 folders and small text files. The number of items per folder is limited to a reasonable size. Each file gets a unique string of around 20 characters. ILV or MFRFixup is required in the package to allow for these file writes.

The Automation Scripts:

A set of PowerShell scripts automate the testing.

RunTest.ps1 will execute a set of runs for a particular test workload – it is not called directly but will be called by the other scripts. It will wait for the system to settle, then execute a loop of test runs for the test workload with an additional pause between runs. There is also logic to perform cleanup between runs when a specific test workload run produces a dirty environment. For a given test run, the script will record the start and end time (delta) of each application[10] run. At the end of the script run, it produces information summarizing the timing results including Minimum, Average, and an average calculation that excludes the largest 10% as impacted by background noise. Results are shown on screen but also saved to CSV file.

RunWpfNativeTests.ps1 will call RunTest.ps1 for each of the test workloads, asking it to run the native application (exe not in the container).

RunWpfPackagedTests.ps1 will first install the basic MSIX package, the one that does not have any PSF components. Then it will call RunTest.ps1 for each of the test workloads, asking it to run the packaged application exe in the container.

RunWpfPackagedLauncherTests.ps1 will first install the MSIX package that includes the PsfLauncher without additional fixups. Then it will call RunTest.ps1 for each of the test workloads. Rather than start the application directly, it will start PsfLauncher to start the application in the container.

RunWpfPackagedFullPsfTests.ps1 will first install the MSIX package that includes the PsfLauncher and all of the primary fixups (MfrFixup, RegLegacyFixup, DynamicLibraryFixup, and EnVVar Fixup). Then it will call RunTest.ps1 for each of the test workloads. Rather than start the application directly, it will start PsfLauncher to start the application in the container.

The four main scripts will each call RunTest.ps1 for each of the test workloads, asking it to run the currently pre-installed MSIX package. Each main script passes variables for the number of repetitions and amount of sleep for initial settling and between repetitions. They also pass variables that allow for cleaning up the package and reporting out data to csv files.

---

[10] In scenarios involving the PSF, this means we are measuring the time to start the PsfLauncher, which then runs the application and then shuts down. The TimMangan fork of the PSF version 2026.07.01 was used.

The OS Bug workaround:

In performing these tests using MSIX packages, I uncovered an issue in the OS. We repeat a given test many times (we had to limit this to 500 due to this issue) and for some of the tests we need to clean-up the package between interactions. For example, if the test creates 50,000 items in a run, we want the next run to start without those already in place. In those tests, when performed on a packaged application we use the Reset-MsixPackage command a memory leak in the OS was discovered. The issue I reported to Microsoft was that this caused a memory leak of Committed memory in a background SVCHost process and also the user's explorer process that runs their desktop. The leak would cause the OS to run out of virtual memory causing crashes in a larger run. I originally intended 1000 repetitions, but found that I could achieve 500 when I monitored virtual memory and manually terminated the svchost process associated with the CLIPSvc service when necessary. An attempt to work around the problem by replacing the reset cmdlet with a remove/add also caused the issue to occur. Eventually, I found that by killing the CLIPSvc process three times it would not restart and the memory leak was manageable. There remains a small leak in the state repository service but it is far more manageable. (A small committed memory leak in the user's explorer process when running packaged apps was also detected, but it did not affect performance of these tests.)

The CLIPSvc is known to be involved with detecting if store apps are licensed and/or need updates. With sideloaded apps we didn't need that. Some basic package testing showed that until we start running out of virtual memory, the results were the same with or without this service running.

The Analysis:

For a given repetition of a test involving a workload in a scenario, there are choices on how to interpret the results of the multiple runs. Generally, we could pick one of the following:

- The average (mean or medium) of the set.
- The average (mean or medium) of the set excluding the first run, or possibly excluding "unexpected" values indicating large background interference.
- The minimum value of the set.

The test scripts produce calculations for all three, however, as discussed in the addendum, we choose to rely on the minimum value to improve assessment accuracy.

The graphs shown in the paper for various scenarios often show both total time of the run in one graph, and the calculation using relative results. In the no-workload scenario that means relative to the unpackaged case. In the workload scenarios that usually means relative to the no-workload case within the same scenario.

The Test Result Scenarios raw results

The raw test results are shown here. Explanations and analysis will follow in subsequent sections of the document. Each scenario was run 500 times. All numbers are in MS. The columns show the total time of a run of an application, except for the last column which shows the time per work item.

Scenario: Application run Natively without a container

#	Workload Description	Minimum	90% Avg	Per Item
0	Baseline without load	1020.932	1023.5641	n/a
1	Sleep for total of 5 seconds	6027.273	6069.1235	50.063413
2	Registry Enumerate 50k items	1019.315	1023.399	0.0
3	Reg Write 50k new items	1020.553	1023.486	0.0
4	Reg Write/Enum 50k items	1033.217	1023.9875	0.000026
5	File Enumerate 50k items	1014.288	1017.2844	0.0
6	File Write 50k Items	4047.166	4952.939	0.0605247

Scenario: Application run in MSIX container without PSF with ILV

Requiring redirection on #6

#	Workload Description	Minimum	90% Avg	Per Item
0	Baseline without load	1049.7	1056.3456	
1	Sleep for total of 5 seconds	6083.476	6117.7317	50.337762
2	Registry Enumerate 50k items	1049.704	1058.4805	0.0000001
3	Reg Write 50k new items	1099.272	1115.2711	0.0009915
4	Reg Write/Enum 50k items	1096.676	1113.4544	0.0009395
5	File Enumerate 50k items	1044.664	1061.7965	0.0
6	File Write 50k Items	16261.3	17367.867	0.2042320

Note: File write 50k items without redirection was 4115.297

Scenario: Application run in MSIX container including PSF Launcher only and ILV

Requiring redirection on #6

#	Workload Description	Minimum	90% Avg	Per Item
0	Baseline without load	1046.607	1065.9664	
1	Sleep for total of 5 seconds	6087.785	6126.2213	50.4118
2	Registry Enumerate 50k items	1048.993	1065.7277	0.0000477
3	Reg Write 50k new items	1125.627	1147.3258	0.0015801
4	Reg Write/Enum 50k items	1127.627	1141.3258	0.00161146
5	File Enumerate 50k items	1045.907	1060.6275	0.0
6	File Write 50k Items	16269.69	17036.6864	0.30446173

Scenario: Application run in MSIX container including PSF Launcher and fixups

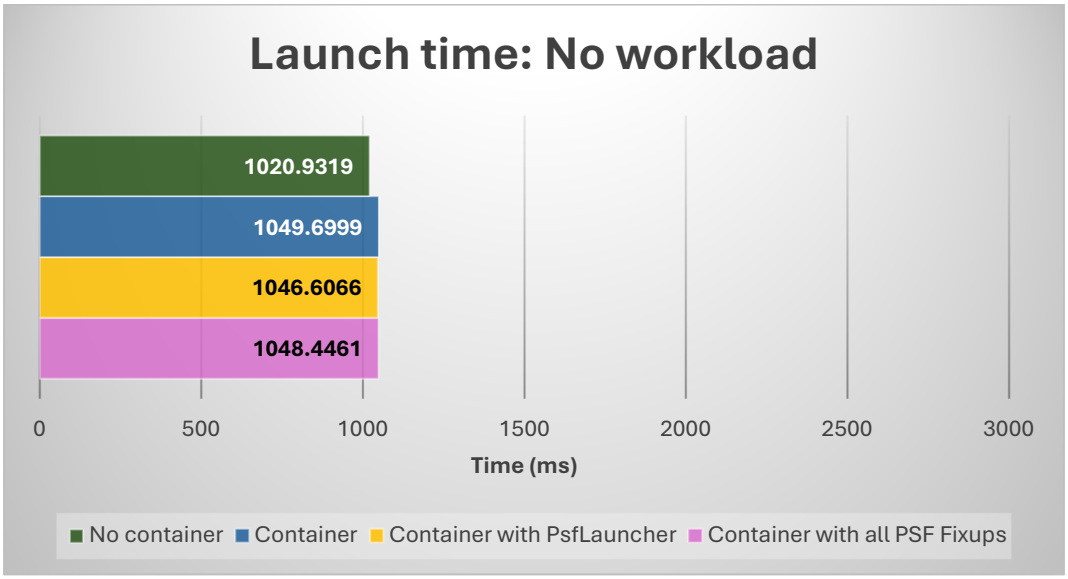
#	Workload Description	Minimum	90% Avg	Per Item
0	Baseline without load	1048.446	1056.7624	
1	Sleep for total of 5 seconds	6064.725	6114.2938	50.16791
2	Registry Enumerate 50k items	1046.508	1056.9734	0.0
3	Reg Write 50k new items	1125.34	1142.6178	0.0015379
4	Reg Write/Enum 50k items	1124.961	1142.3741	0.0015373
5	File Enumerate 50k items	1047.437	1056.4998	0.0
6	File Write 50k Items	21336.04	22412.881	0.4057518

Analysis

The test app consists of two primary threads, the GUI window that appears and a background thread to process the workload. In all test cases, the background thread will update a progress bar from 0-100 with 100 signals to the main GUI to update the display.

Launch Performance

The best test of launch time is made by comparing the first test case in each of the four scenarios. This first test case is the case of the application with no additional workload. Although the test is actually a full life of the run, the shutdown time of the app should not be significantly different between scenarios. The test application contains minimal application-specific startup code as we want to separate launch performance from that of runtime activities. The background workload in this test case only consists of signaling to the GUI to update the progress bar (to be consistent with other test cases).



All test results shown here represent the externally measured time from start to finish. For the first two scenarios this means the application, and for the last two this means for PsfLauncher, which then runs the application and closes after the application closes.

Native vs Package (No PSF): From this result we can see that placing the application inside the MSIX container does have a small amount of overhead, but not enough to cause a human noticeable delay in the launch. The small delay measured would come from additional time by the OS to set up the container, and the appx svchost process checking to see if the application launch should check for updates (the package was generated without update checking requests).

I did not test a larger application. But while the total time to run a larger application may be bigger, assuming there is not a lot of added application initialization in the larger app, these causes should probably not significantly increase the difference in this startup. Feel free to verify that assumption yourself.

Vs PsfLauncher. When adding the PsfLauncher, we expected the possibility of an additional process launch to be measured (the PsfLauncher must start up and then it starts the application and waits for it to terminate before shutting itself down – time measured case in this scenario is for the lifespan of the PsfLauncher. The results were so good, seemingly identical to the packaged scenario without the PSF that a lot of verification went into confirming the result. This indicates that the addition of PsfLauncher alone does not have a significant performance impact.

Vs Full PSD. When the PSF Fixups are loaded, there is a small amount overhead. The PsfLauncher suspends the application to allow for injection of these dlls and for the dlls to initialize before resuming the application. I expected that we might find significant overhead here as all work on the PSF to date has been about application compatibility. From the PSF Code, there would be two things of potential concern:

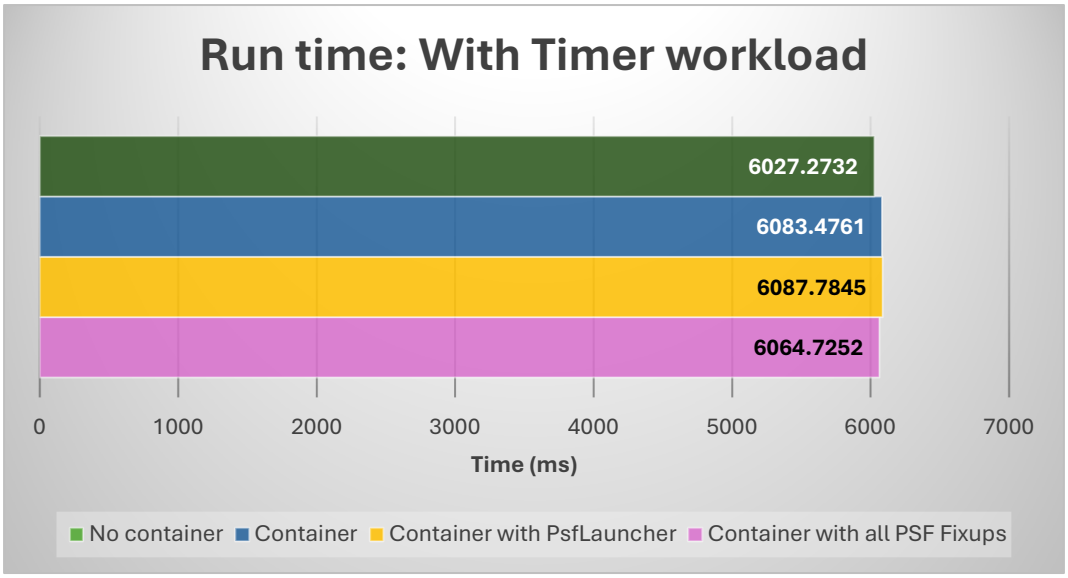
- There is a design consideration in some of the PSF fixup components – especially the MfrFixup which provides file assistance – which favors taking time up front during fixup initialization in order to optimize runtime performance. The results indicate that their may be room for improved optimization here, especially as it comes to the order of fixup initializations. This will be investigated later this year.
- The PSF contains logging to the Windows Debug Console port by default. This is configurable, but the default setting which logs during process launch only was enabled. The overhead for this is small normally, but in the presence of any software reading that port (Like a debugger or DebugView) the delays are more noticeable. Anyone testing performance of the PSF should either disable the PSF logging (set the logging level to 0) or avoid any software from reading the port.

In the test we kept the default logging level of 2 (logging of process start/stop activity only). The result shows a negligible difference between the launcher along or bringing in those fixups (far less than the margin of error in test results).

From testing this scenario, I conclude that there is no significant penalty to the launch time of an application due to running it in a container, even with the full PSF involved. If real-world applications show a performance penalty, it is likely due to application-specific initialization. The overhead of that initialization should be the same as that of the runtime workloads that are the target of the additional scenarios.

Timer Workload

This scenario was created simply to validate the testing rig. If I add a workload of a 5 second sleep, the resulting time should be 5 seconds more.



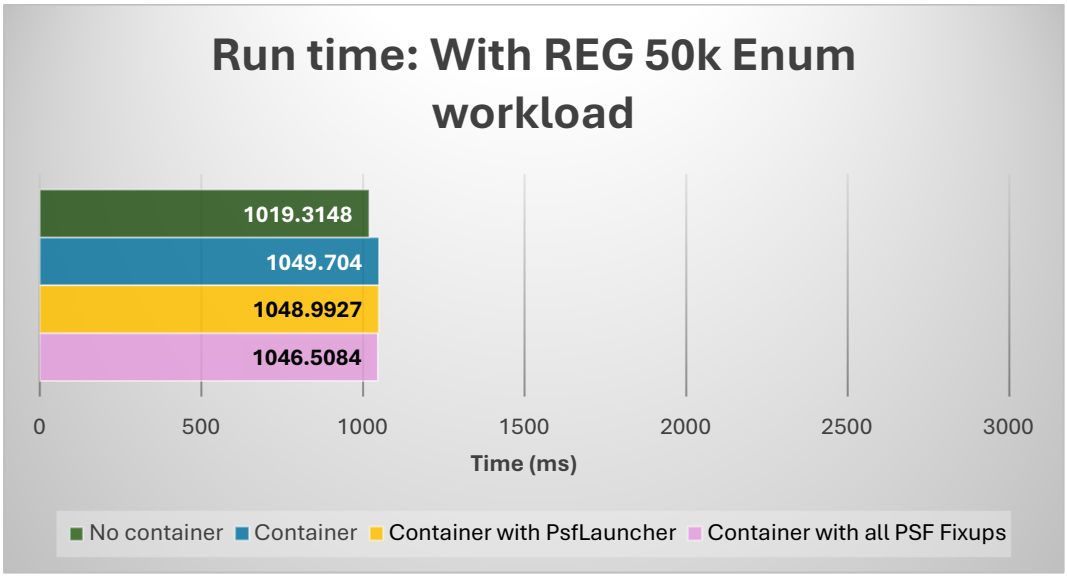
The times shown here are total time, not time relative to the non-workload case.

Comparing this to the first test scenario, we see that the workload added slightly more than 5 seconds over the native run without the workload. The conclusion of a sleep call is variably later than requested due to the dependence on a 15 ms hardware interrupt and potential competing CPU needs at the time that expiration is detected . Due to the nature of how Windows works, timers can take longer than requested, an average of 60 ms in testing performed outside of this paper[11]. This overhead of a timer workload was reasonably consistent both inside and outside of the container.

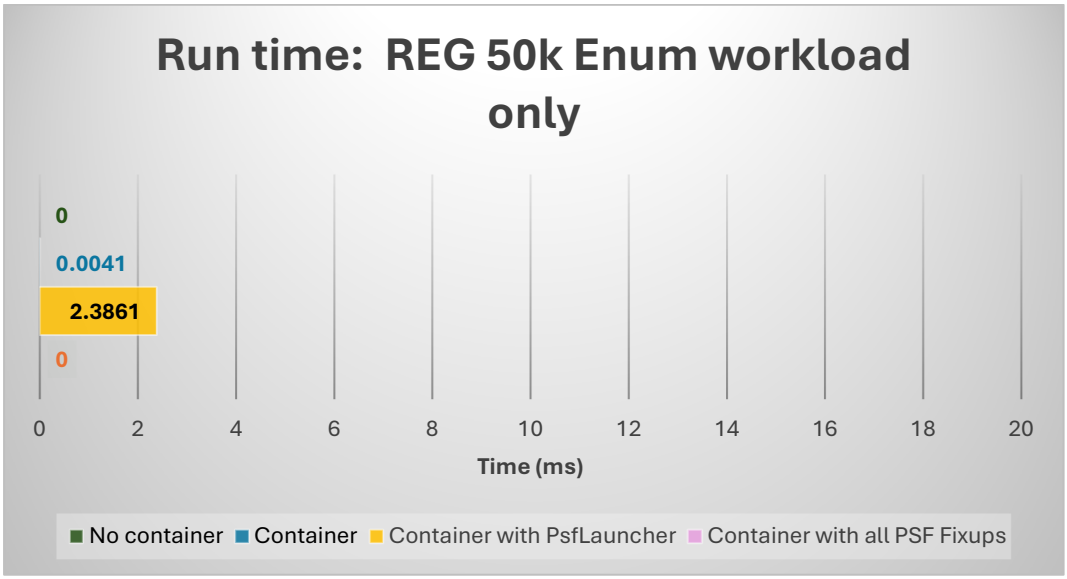
[11] I broke the 5 second timer into a serialized set of 100 timers of 50ms each. This added 6 seconds to the runtime instead of 5, consistently both inside and outside the container.

Registry Enumeration Workload

In this test, we look at the impact of enumerating registry keys and items that exist in the native Windows registry. For simplicity, the packages do not include a virtual registry.dat file, which means in all cases for this workload the enumeration is falling through to the native registry (HKCU\Software).



The times shown above are total time, not time relative to the non-workload case. The following graph shows the times relative to the non-workload case of the same scenario.



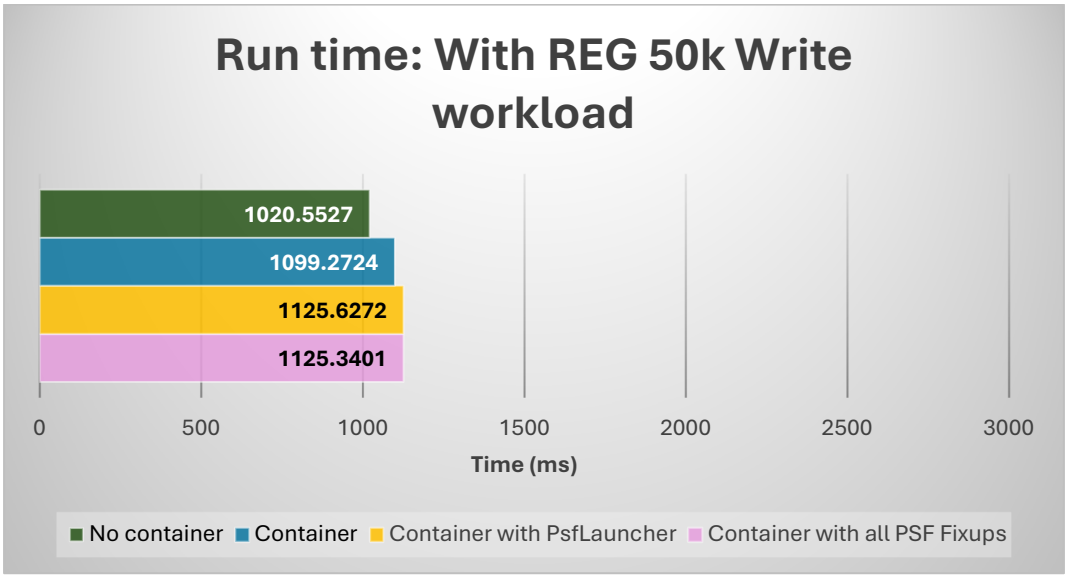
Remember that the test results, even with the minimum selection of 500 runs, is still only accurate within a handful of ms. Even with the worst-case number for the full PSF scenario that 2.4ms value works out to about 0.00005 ms per enumeration call.

Registry enumeration is clearly not a performance issue in any scenario.

If we had seen a significant performance difference, additional registry enumeration and reading test scenarios would have been warranted, including enumerating non-existent keys (a performance issue we saw in App-V), querying the values of items, and also testing against a large virtual registry that pre-existed in the package. The development of those tests is left to the reader.

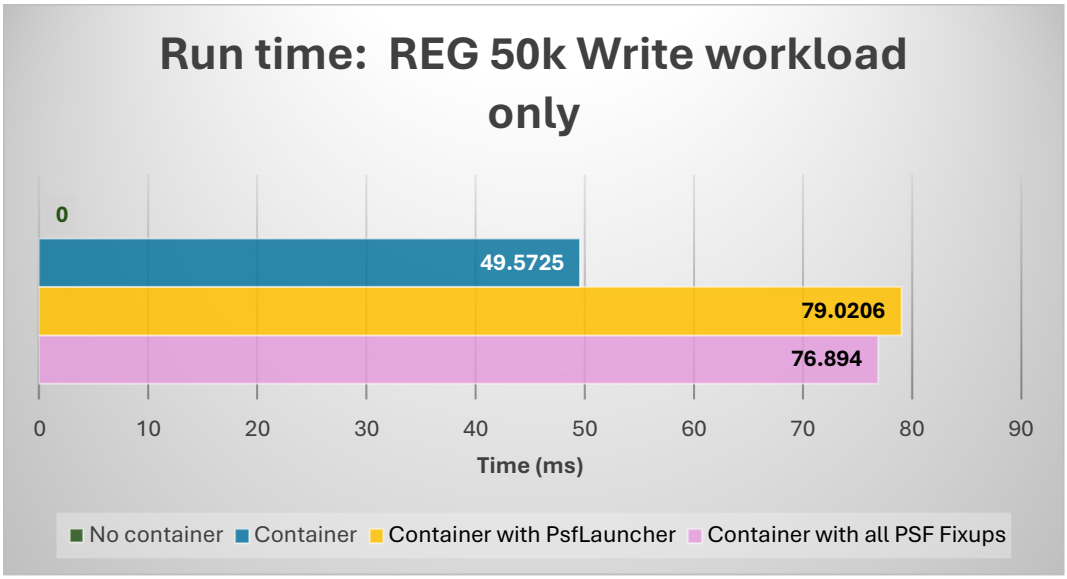
Registry Create/Write Workload

In this test, we look at the performance of creating registry keys and values. In the native scenario, these are being created in the real registry, in all of the packaged scenarios they are being created in the virtual registry. 50,000 creations are made under HKCU\Software, broken up into groups to prevent too many items being under a given key.



The times shown above are total time, not time relative to the non-workload case.

The results appear to show a small bit of overhead when run in a package with or without the PSF, but not much.



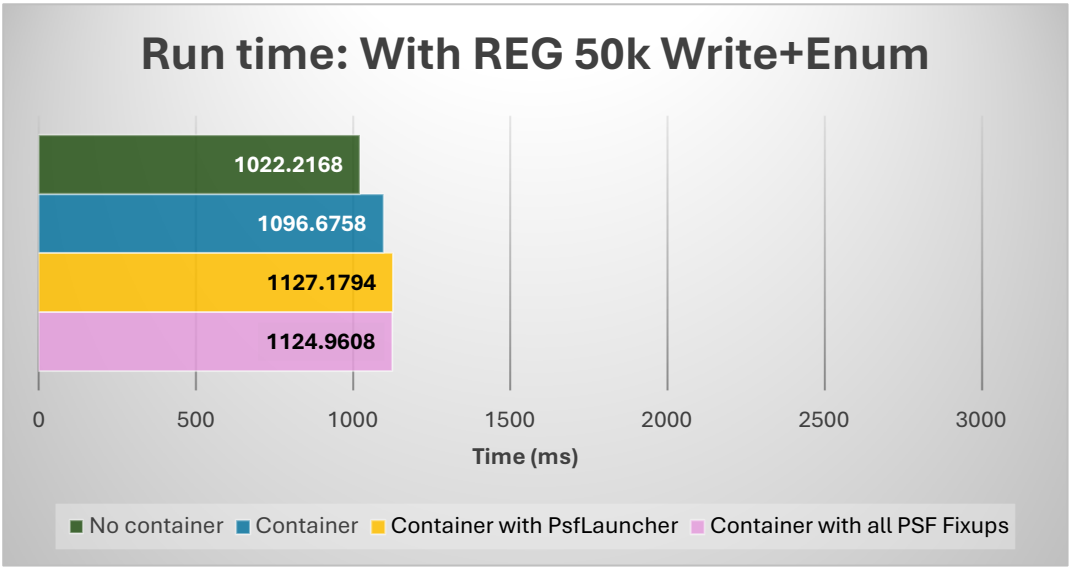
This graph shows the times relative to the non-workload case of the same scenario.

There does appear to be a small amount of overhead to writing to the virtual registry in all packaged scenarios, likely from having to check multiple locations. Although small enough that the end user would not notice, it probably exists.

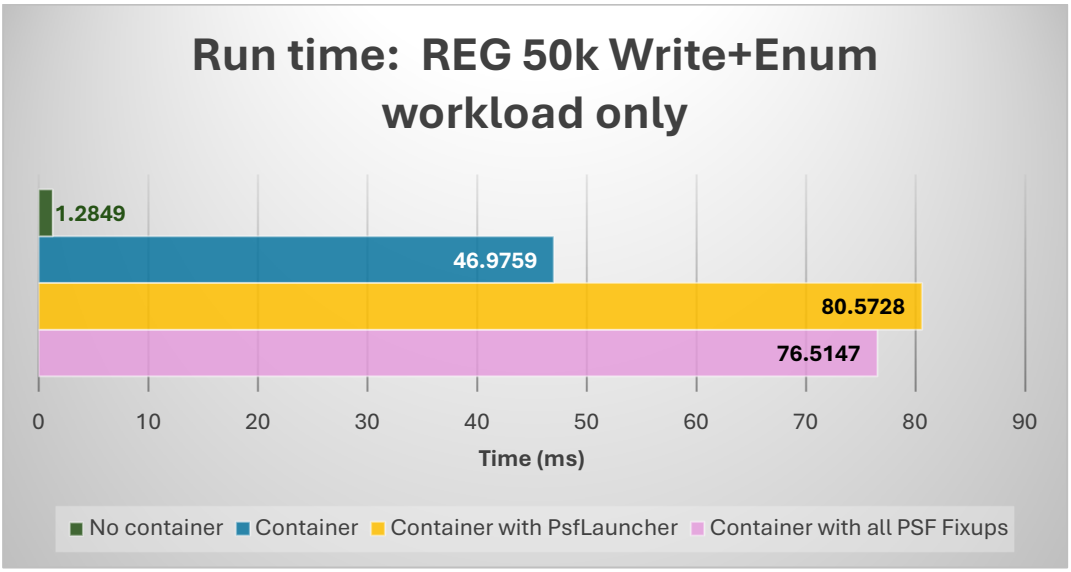
In the worst-case scenario of the package with the full PSF, that amounts to 0.0015 ms per write.

Registry Write + Enum Workload

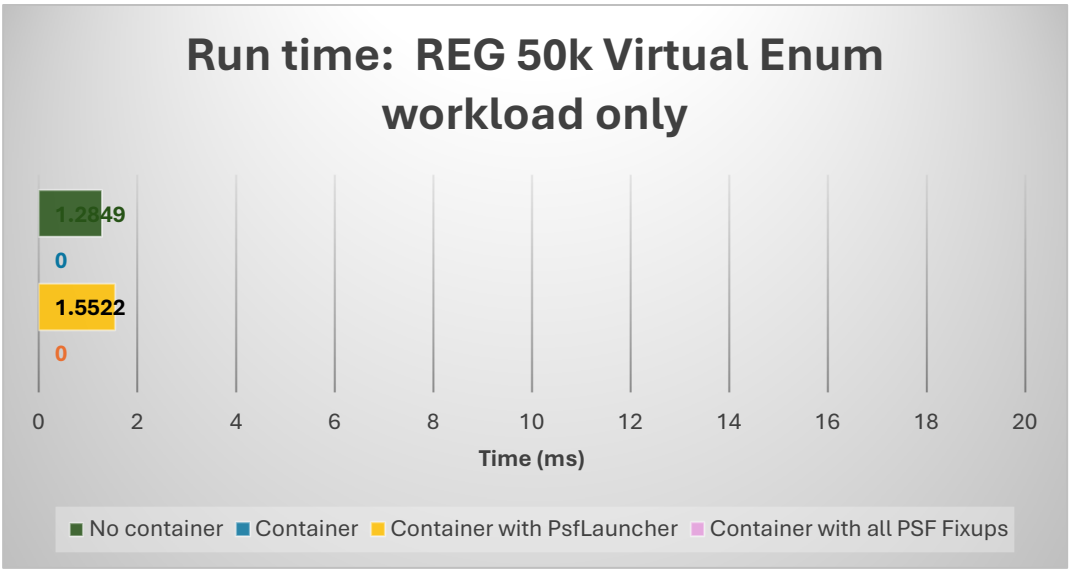
In this test, we look at the impact of enumerating registry keys from the registry in a scenario where the packaged version is enumerating the virtual registry items rather than native items. For simplicity, the packages do not include a virtual registry.dat file, and we first create them in this test, then after all have been created, enumerate them. We can compare this against the write-only scenario in an attempt to extract the virtual enumeration impact. In the unpackaged case, the real registry is being used. In all cases the HKCU\Software key is being used.



Times shown in the graph above are total time of the run. The graph below shows the time relative to the no workload test of the same scenario.



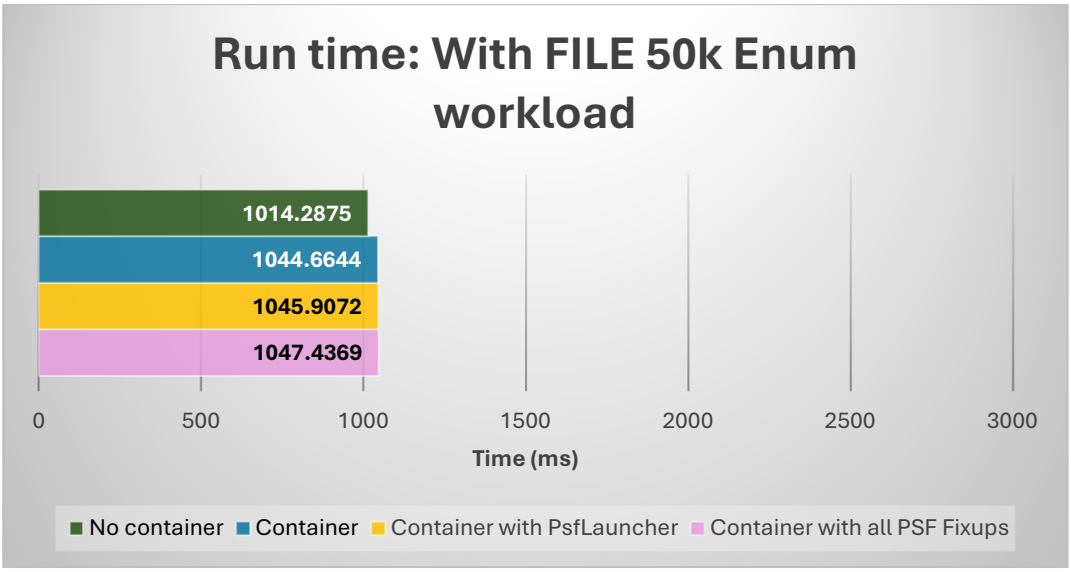
The following graph shows the time relative to the Reg Write scenario to provide an estimate of the time to enumerate from the virtual registry.



The results of the test indicate that reading from the virtual registry has a insignificant effect on performance.

File Enum Workload

In this test, we look at the impact of enumerating folders and file in the different scenarios. For simplicity, the packages do not include a large set of files, so the enumeration is that of the C: drive, not files in the package.

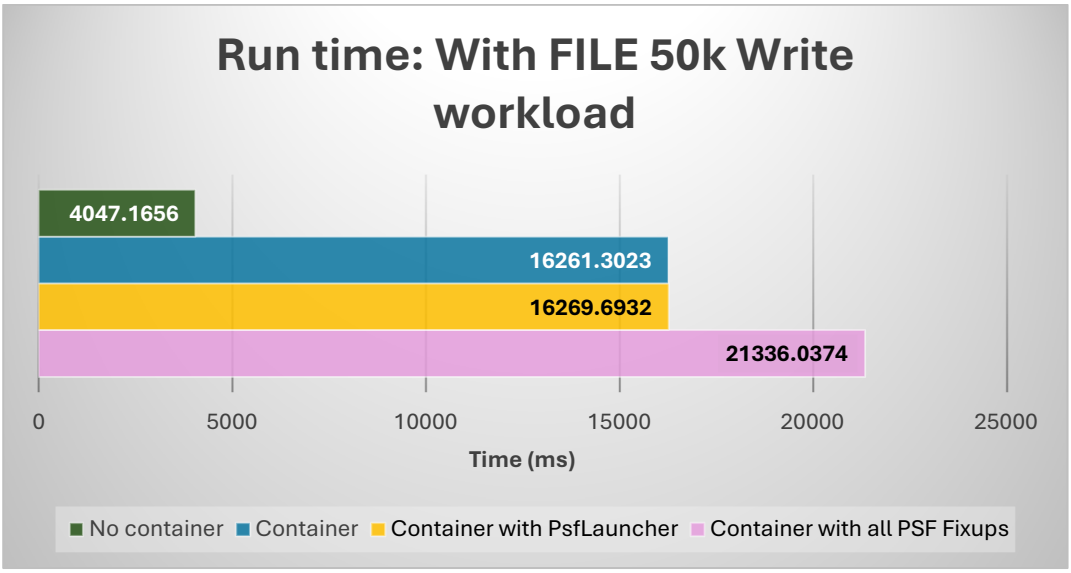


The numbers shown in the graph above are the full time of the application run.

This test shows insignificant differences, even in the full PSF Fixup scenario where three different locations (Package saved data, package, native) must be consulted on each enumeration.

File Write Workload

In this test, we look at the performance of creating file system folders and files containing data. The files are created in a subfolder where the application executable lives. In the native scenario, these are being created in the real file system, in all of the packaged scenarios they are being created in the user redirected area. 50,000 creations are made, broken up into groups to prevent too many items being under a given folder. The size of each file is quite small (a unique text string identifying the file).



The numbers shown in the graph above are the full time of the application run with the process exclusion present in Windows Defender[12], and with InstalledLocationVirtualization enabled in the packaged app without PSF scenario and the PsfLauncher only scenario to enable the files to be written. The last item which is using the PSF with MfrFixup without ILV. These tests were with the files being written to the file path of the application, and therefore being redirected.

Not shown here is results from testing where the files are written in areas without redirection. In all cases, those results look only slightly longer than the native application, much like the no workload case.

That the result of the native time being longer than that of running inside the container without the PSF, or with Psf Launcher only, is due to the use of InstalledLocationVirtualization and the redirection. Keep in mind that apps don't normally create 50,000 tiny files. It is believed that the redirection overhead is due to having to

[12] The minimum value for the native case ballooned to 67seconds with Defender scanning the process in the Native scenario. The other scenarios were not affected. Other AV vendors were not tested and might impose different performance impacts.

check multiple locations for creating the file and that actual writing of file contents are not affected by ILV or MfrFixup once the file is open.

The test result of the full PSF is not unexpected as performance work on the MfrFixup is likely needed. Applications that create a lot of files would have a noticeable performance impact.

Addendum

Why 2 test runs will never produce the same results

The Windows Operating system architecture relies down at the hardware level on a timer tick that interrupts the processor at a rate of 15ms between ticks. While there is a high performing clock that can be queried for high resolution time values, that requires the querying software to be present in a CPU thread.

The 15ms tick dates back to the earliest days of DOS, it was part of the original IBM architecture and it remains in place today. (I even wrote a paper in 2004 suggesting that the industry look at changing this [13]).

Many operations taken by applications cause the application thread to voluntarily give up the CPU, waiting on an event to complete. Examples include waiting on a file IO operation, or even waiting on a timer.

In test workload 2 in our app, the application asks to sleep for 5 seconds. In a variation of that test we instead replaced that with a 50ms sleep that was repeated 100 times. In that case, the windows API sets a 50ms timer, creating a timer object scheduled to complete in 50ms, then the application thread waits on that timer to complete.

The OS cannot detect that the timer completed until the next 15ms hardware tick occurs after the time is up. So even in a completely idle system with lots of CPU threads available, that sleep might take anywhere from 50 to 65 ms depending on when the sleep was requested versus when the next tick occurs. When that tick occurs, the kernel will determine that the timer expired and place the application thread into the ready queue. On that idle system, the thread will immediately be placed into a CPU and resumed.

But if the system has other activity to perform, then the scheduling depends on the relative priorities of other threads that want to run. Because our application thread had been waiting, it is automatically given a temporary priority boost of 2, which should improve its odds, but keep in mind that any other process threads likely were recently waiting on an object and also have the same boost. In a completely CPU bound scenario, it could take 150 ms to get scheduled (there are rare situations where it could be 1 second, but then your system is probably toast already).

Therefore any workload that waits on object completion will have a random about of additional delay on each run.

---

[13] **It's Time to Change the Timers** https://tmurgent.com/appv/en/resources/white-paperlist/84-it-s-time-to-change-the-timers

As seen in the published result, adding a 5 second wait as the workload added about 5 seconds to the run, but when we tried it with 50ms sleep repeated 100 times it added over 6 seconds. Each of those 50ms sleeps took an average of more than 60ms.

This is why the results can vary from one run to another no matter how hard we try. We just have to account for the noise when looking at performance.

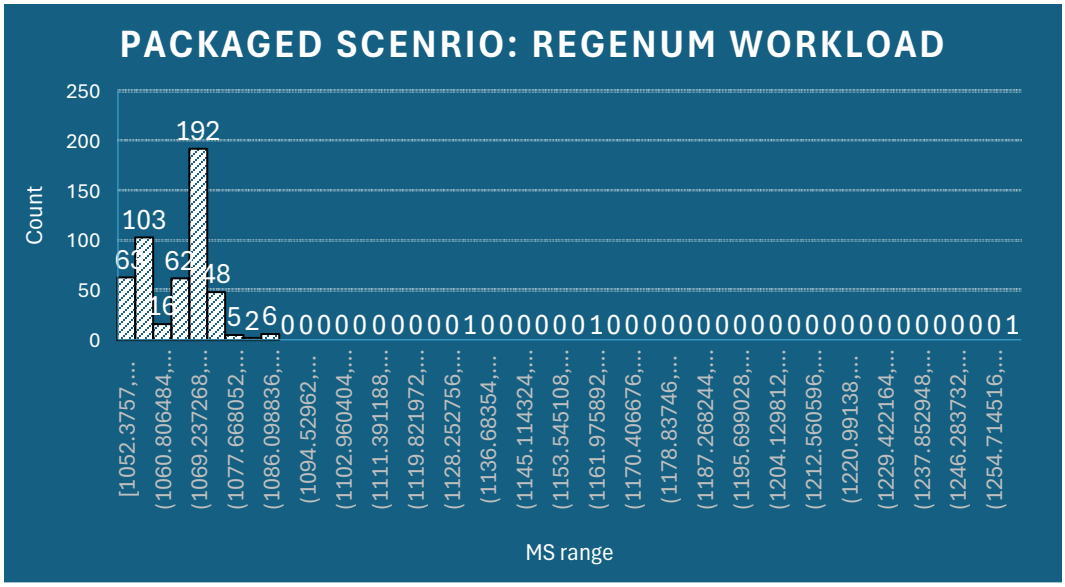
On using Minumum value

For a sufficiently large set, the difference between mean and medium averages should be insignificant, and if you care to debate which is better feel free but keep us out of it.

We certainly want to exclude the first run, as previously explained. For practitioners looking at typical application performance, the choice seems of excluding obvious outliers is obvious, but what makes an outlier is more difficult to automate.

But for our testing looking into causes of performance, selecting the minimum value is more valuable. The minimum value in this case represents the best-case scenario where the background activity of windows had the least amount of impact on the test run. This minimum value provides us with the best value of how long that test run took to perform in that environment.

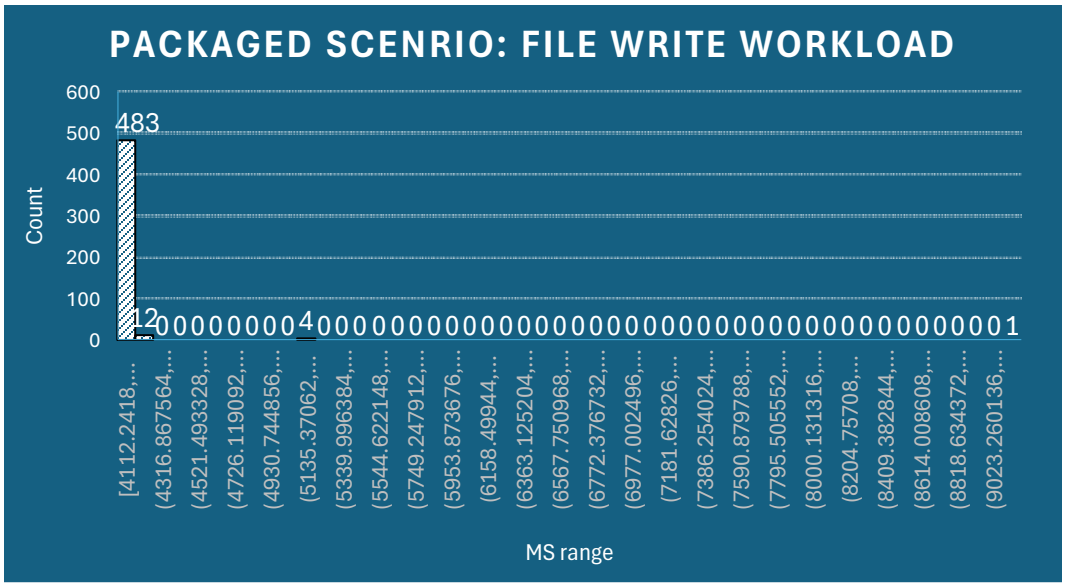
Here is a histogram that shows the count of how many results fell into certain measured ranges in a 500 repetition test.



This was a short duration test but you can see the pattern that often occurs when looking at results this way. There is a minimum value, and then peaks that occur

20ms away. Because this was a short test with a compact workload we only see one peak, but in different workloads we sometimes see 2 or three peaks like this.

Here is another example.



We often also see events in the long tail that skew the average, and these are the events that we might not see if the test repetition was repeated. That one event added almost 5 seconds to the run. This was probably a case of the OS waking up to do something more intensive, and at a higher priority. Perhaps the AV agent updated its malware definitions and froze file operations while it replaced it. Or something else. This is why if averages are to be used, excluding the outliers is a must. But we aren't measuring for end user performance in these tests, the goal is to shed light on system components that may be the cause of performance degradation. And the minimum value best represents the work requested by the workload.

Background activities in windows also affect the minimum value when a set of 500 repetitions is run a second time. For the test results presented in this paper, additional tests (not shown) show that the difference for the minimum value in a 500 repetition test will vary by 1ms, while the average may vary, maybe by 20ms of variation on a 1 second test. And as these interruptions are serial, a longer duration test will have even greater average result variation. Use of minimum value versus average does not affect the outcome of the analysis, only the confidence level that it was not caused by a bad test run.

Increasing the repetition count would reduce the measurement error significantly further. Unfortunately, we were limited to a 500 count by an OS bug in Windows at the time of this testing.