

Effects of “Fonts” in App-V 5 SP2 with HotFix4 Deployment Performance

TMurgent App-V Performance Research Series

Can you REad THis?

June, 2014

Table of Contents

1	Introduction.....	4
2	Background on Deployment Performance	5
3	Background on Windows and Fonts	6
3.1	Natively Installed Fonts and Performance	7
3.2	Windows 7 and Font Limits	8
3.3	App-V 5 and the Font Extension / Subsystem	8
3.4	App-V 5 and Font Limits.....	9
3.5	Detecting Fonts inside a Package	9
4	Summary of Where Impacts of Virtual Fonts Are Felt	11
4.1	Sequencer Limitation with Fonts.....	11
4.2	Working around the Windows Font Limitation.....	11
4.3	Standard Scenario Testing results.....	12
4.4	Results Interpretation	13
4.5	Improvement Options.....	15
5	Testing Strategy Used	16
5.1	About the Testing Platform	16
5.2	About Test Packages and “Streaming Configuration”	16
5.3	About the Testing Methods	16
5.3.1	Test Package.....	17
5.3.2	Test Pass.....	17
5.3.3	Test Cycle	18
5.4	About the Test Results Accuracy.....	18
6	Test Packages Utilized.....	19
6.1	Lots_OfNothing (Warmup)	19
6.2	Lots_OfFonts_RenamedPVAD	20
6.3	Lots_OfFonts_RenamedVFS	21
6.4	Lots_OfFonts_CopiedPVAD.....	21
6.5	Lots_OfFonts_CopiedVFS.....	22
6.6	Lots_OfFonts_Installed	22
7	Detail Test Results.....	23
7.1	SCS Mode Without Mounting.....	24
7.2	SCS Mode Testing with Mounting	26

7.3	SCS Mode Disabled, No Mounting	28
7.4	SCS Mode Disabled, Mounting Used.....	29
8	About This Research Paper Series.....	30

1 Introduction

The purpose of this research paper is to document the effects that fonts have in Microsoft App-V Virtual Application Packages. But along the way we get to learn a bunch of interesting things about fonts.

The effort is squarely aimed at answering questions on how the addition of Fonts in a virtual application package affect performance.

This work is part of a series of efforts to characterize the impact that different application elements have on the performance of virtual applications under App-V 5.

Most readers of this research will find themselves satisfied with reading the second and third section of this paper. The remaining sections detail the testing process, packages used, and provide further test details and additional findings.

2 Background on Deployment Performance

Being the first of the 8 papers to really deal with deployment performance (the BigFile paper was more about runtime performance), I should spend a little time talking about deployment performance in general.

Moving from App-V 4.* to App-V 5 originally caused a lot of enterprises heartaches over the slower deployment times. Especially in a non-persistent VDI scenario, the time to complete package deployment (usually called “publishing time” by most people, but since “publish” is one of the steps I prefer to call it “deployment time”) was just too long in App-V 5.0.

Microsoft made a few adjustments in the App-V 5.0 SP2 release, but it wasn’t until Hot Fix 4 for SP2 that they made significant progress. While I have not made comparison tests myself, the consensus seems to be that with the Hot Fix 5.0 is generally as good or better than 4.6 when it comes to deployment time. But it depends on the package, and we can always use better.

What consists of deployment time can vary from one user to another. It depends mostly on the environment that they receive the applications.

Those with non-persistent (or semi-persistent) desktops are mostly concerned with the time to add and publish the package. Assuming that Shared Content Store Mode (SCS) is in use (likely in non-persistent VDI and possible with RDS/XenApp implementations), first time launch performance is also very important to them.

But those with persistent desktops generally are more concerned with launch time of the virtual package than deployment time which only happens once.

The tests in this series use packages that are created without training the publishing streaming block. They are tested with SCS enabled and disabled, and with and without mounting.

Normally, with SCS enabled, no mounting is performed. But you can train the sequence to include files in the publishing block to ensure that they are cached locally. Plus, in a pinch you can script a mount command of a particularly important package or include in the OS disk image, to improve the performance.

Normally, with SCS disabled, you would have AutoLoad enabled to perform background streaming to the local cache. These tests run with AutoLoad disabled so that you can see the performance when the user doesn’t wait long enough for AutoLoad to complete.

3 Background on Windows and Fonts

Typography and the design of typefaces used for printing, is an ancient art-form. Wooden blocks with hand-carved letters created by Calligraphers were used for printing at least as far back as the Han Dynasty (206AD). Western typography really gets going around 1440 when Gutenberg invented the first western printing press using movable type fonts. Today, the use of a particular typeface is often used to emphasize branding and style by people and companies. The computer has embraced this concept with fonts, which are software file definitions characterizing a typeface by defining a large set of standard parameters that allow the computer to produce the typeface, both on-screen and to printers to produce hard copy.

Initially, the computer operating system provided one (or more) fonts for screen display and that was all you got. When technology advanced enough to allow additional fonts to be added on demand, the fonts made available by third parties followed the licensing model for physical typesets for printing and required licensing of the fonts themselves.

One of the most popular providers of loadable fonts was from Adobe Systems. These fonts used a format for the font definition called ATM (Adobe Type Management) and ATM fonts were licensed by a large number of companies. These fonts are generally licensed on a per-user basis, which leads to interesting situations involving Remote Desktop Services (formerly Terminal Services) scenarios. There are still quite a few companies continuing to use some per-user licensed ATM fonts today. Traditionally, App-V has been used in these situations to limit font exposure to a subset of users in these situations by virtualizing the font with the application that requires it – and then limiting access to the application by those with a license to use the font.

Microsoft, for their part, developed a secondary format and built support for it inside the OS. This format, called True Type Font, allowed for a single font file to be used to depict the font at a variety of scales. Whereas you needed a different ATM font definition file for each point size, a single TTF font file could render multiple point sizes. Microsoft also released a number of fonts with the operating system itself. Third parties have also produced thousands of additional TTF based fonts, many of which are free to be installed on Microsoft operating systems.

Companies often standardize on a small number of fonts for use in all of their external communications, as part of their branding. There are applications and situations, however, demanding a large pallet of fonts be available to the user. AutoCAD 201, for example, 5 installs over 100 fonts. But the need for even larger numbers of fonts is especially true for people creating advertising or other forms of media for others. We often think of these jobs as using Apple computers rather than Windows, but now I think I know why. But we'll get t that later.

3.1 Natively Installed Fonts and Performance

While fonts are not generally considered to be a major contributor to poor performance, the impact of adding a large number of fonts can be measured. And you can run a simple test to see this effect.

When a font is installed, the font file itself is copied to a specified location, and registry entries are added so that it may be found. When an application wants to use a font, it generally does not know if that particular font file will be present. Microsoft designed the API for developers to help solve this. The developer (generally) does not ask for the specific font file. Instead, the developer makes an API request filling out the parameters that define the font they want. The developer may be as detailed in the request as they want. If they want a particular font, they fill out all of the detail matching the parameters for that font. The API will perform a search on all of the installed fonts and return the font that best matches the requested font. So if the developer asks for a 12 point font in the Times Roman family with serifs, and for some unknown reason someone had uninstalled that standard font from a system, the API will return the next best thing (for which most of us would not notice).

A key point here is that when an application installs a font, it installs it to the system, making it available to any application looking for a font to use. So whenever any app looks for the font it wants, the library call takes a bit longer to return the best matching font.

To see the effect of having a large number of fonts on a system, you would perform the following test on a system. The system requires a program that shows you all of the fonts, similar to the way Microsoft Word does when you click on the Font selection pull-down menu. In this case, the program manually reads the registry entries to determine font names, determines the full set of parameters needed to get this font, calls the API to get that font, and then uses that font to display the name in the menu itself.

- Reboot, Login, start Word
- Pull-down the Font selection menu, timing how long it takes to display.
- Install an additional hundred fonts.
- Reboot, Login, start Word
- Pull-down the Font selection menu, timing how long it takes to display.

The difference in time to display the menu will be noticeable to the naked eye. Of course it only works once as Word will keep the font open for re-use.

This test leads me to the first conclusion about fonts and performance:



INSTALLING FONTS NATIVELY TO THE OS HAS A TINY, AND YET MEASUREABLE, NEGATIVE IMPACT ON PERFORMANCE TO MOST EVERY APPLICATION ON THE OS, EVEN IF THE APP DOESN'T WANT TO USE THE ADDITIONAL FONTS.

Because of this effect, virtualizing the font inside the package can be a performance advantage. By keeping applications that do not need the extra fonts from seeing them we speed up requests for fonts, but, this is only an advantage when the virtualizing agent doesn't add too much overhead to handle the virtual fonts. Prior to App-V

5, this was the case, but as you can see in this research paper, the new font subsystem in App-V 5 is more impactful and eliminates this advantage. Still, there are licensing advantages to virtualizing those fonts requiring font licenses.

3.2 Windows 7 and Font Limits

In researching fonts on Windows, I found that there appears to be an undocumented limit on the number of fonts that may be installed. This limit appears to be *around* 1000, at least on my test platform of Windows 7 SP1 x86. There are reports of the limit being much lower than 1000, as low as 400, but my testing as shown a limit around 1000. As the OS comes with about 130 fonts, this effectively limits you to adding about 870 fonts in a single package.

1000 fonts is certainly a lot of fonts, and far more than most people really need. But there are a few large programs that install hundreds of fonts as part of the installation. And certainly someone in the business of advertising or marketing may want a much larger pallet of fonts to work with to perform their daily job. So the 1000 font limit, while unimportant to most of us is run into by some users. This limit may be one of the reasons these people use Apple Computers.

3.3 App-V 5 and the Font Extension / Subsystem

One of the specialized subsystems in App-V 5 is the font subsystem. The sequencer is designed to detect the presence of fonts inside a package and identifies these as Font Extension Points. Unlike many of the App-V 5 extension points, fonts considered "internal" extensions, meaning that only the virtualized application has access to the fonts.

Because of this, fonts affect performance as files inside a package, plus a run-time performance hit when the package is started. This is addition to the small runtime performance hit of a font being available to an application, but at least this hit is now limited to the virtualized application and not all applications on the OS.

I should note that font installation involves both a registry change and a copy of the font file into a sub-folder of the Windows directory. Because of the file copy, the font files themselves will be in the VFS area. As reported in a different research paper in this series, the inclusion of these font files in the VFS cause specific deployment performance changes even if the virtual font sub-system is disabled.

3.4 App-V 5 and Font Limits

The App-V Sequencer automatically detects fonts and produces the list as part of the AppXManifest file, for special processing by the client font subsystem.

I have seen (in 5.0 SP2) that not only does this detection method used in the sequencer detect any installed fonts, it manages to detect all font files inside the package, whether or not the font was actually installed. This is probably a bug in the sequencer logic, as although the additional fonts are processed by the font subsystem at the client, the virtual application still cannot use them without registration. While I normally do not advocate routine “cleanup” of packages in App-C, this leads to finding number 2:

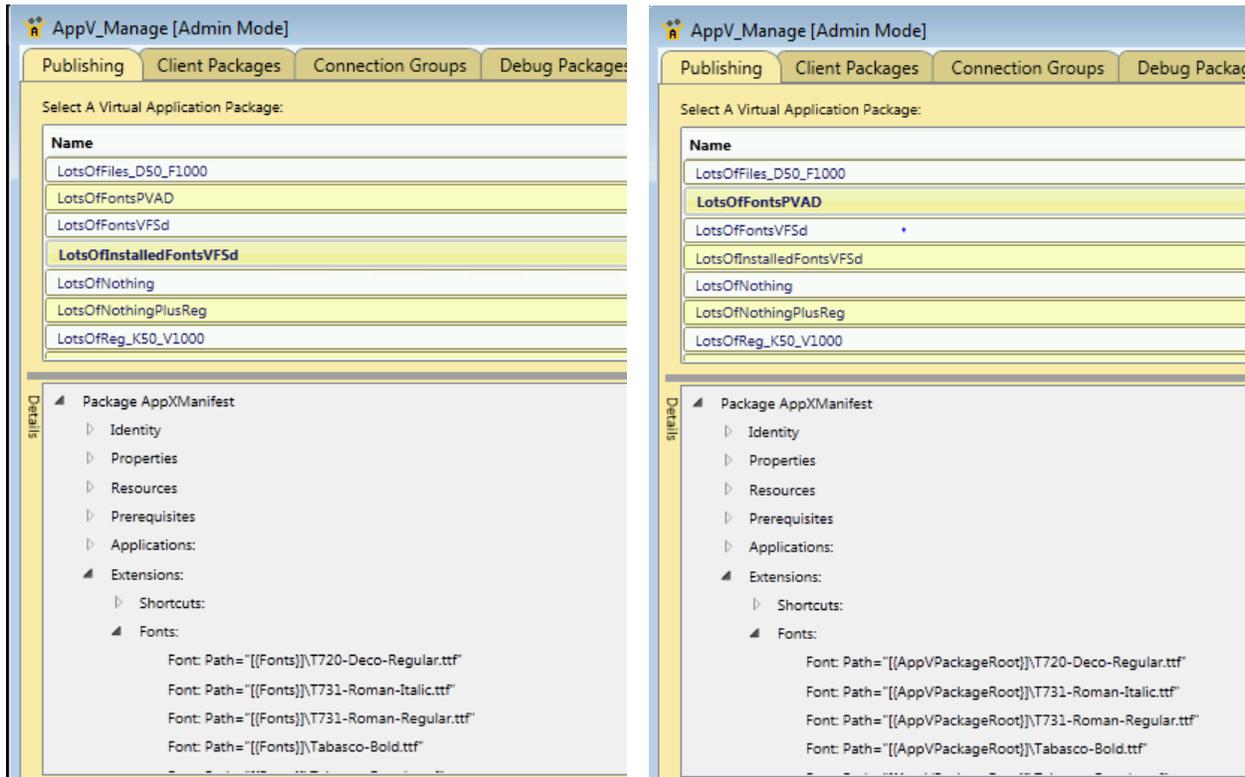


ALTHOUGH IT RARELY HAPPENS, IF ANY FONT FILES ARE IN THE PACKAGE BUT THE FONT IS NOT REGISTERED YOU SHOULD PROBABLY CLEAN THEM OUT FOR OPTIMAL PERFORMANCE.

3.5 Detecting Fonts inside a Package

How do you detect installed and uninstalled fonts in the package? You can look at the path location of FONT files shown in the DeploymentConfig or internal AppXManifest file for the App-V package.

When fonts are installed, they are copied to the Windows/Fonts folder (variabilized as {{Fonts}}), so if the sequencer detects non-installed font files, the virtual font listing in these files will show a different path. The images below are taken from the AppV_Manage tool where you can view the AppXManifest file. The package on the left shows installed font entries, the package on the right shows detected fonts that were not installed:



You can also see a summary of detected fonts using the Analyze button of AppV_Manage:

The screenshot shows the 'Package Stats by AppV_Manage' window. It displays the following information:

- File:** \\tmuhost\Contentold\Test\LotsOfFonts\Lots_OfFonts_5Installed\Lots_OfFonts_5Installed.appv
- PVAD:** C:\Lots_OfFonts_5Installed
- Architecture:** x86/x64
- OS:** Any OS

Archive Size:	33,518,711	# Central Dir Entries:	792	# COM Handlers:	0	# Shell Extensions:	0
CentralDirSize:	89,569	# of Empty Folders:	3	COM Inproc:	0	Column Handlers:	0
BlockMap Size:	65,484	# of Opaque Folders:	2	COM OutOfProc:	0	Context Handlers:	0
StreamMap Size:	231	# of SidebySide Pkgs:	0			Data Object Handlers:	0
AppXManifest Size:	5,198			# AppPaths:	0	Drag&Drop Handlers:	0
Registry Size:	36,852	# of Shortcuts:	2	# AppCapabilities:	0	Drop Target Handlers:	0
FilesystemMeta Size:	9,628	# of Fonts:	770	# Software Clients:	0	Image Handlers:	0
PackageHistory Size:	493	# File Type Assocs:	0	# Environment Vars:	0	InfoTip Handlers:	0
		# URL Assocs:	0	# Browser Helper Objs:	0	PropertySheet Handlers:	0
		# Services:	0	# Active X Controls:	0	Other Handlers:	0

4 Summary of Where Impacts of Virtual Fonts Are Felt

This section highlights the most important results. Additional details appear in subsequent sections, however many readers will stop reading after this section.

4.1 Sequencer Limitation with Fonts

The Sequencer (as of 5.0 SP2) has a bug in that the total number of font files it detects (whether registered or not) is limited to somewhere under 1000, possibly around 800. If you exceed the limit, the sequencer will crash at the end of the configuration phase. This is finding number 3:



THE SEQUENCER WILL CRASH IF YOU ADD TOO MANY FONT FILES IN THE PACKAGE, WHETHER OR NOT THEY ARE INSTALLED. THIS LIMIT IS SOMEWHERE BETWEEN 800 AND 1000.

4.2 Working around the Windows Font Limitation

One aspect of the testing was to determine the maximum number of fonts supported. A single App-V package is limited by the roughly 1000 fonts supported by Windows. Since the OS starts with nearly 130 fonts, this causes a practical limit of about 870 in a single package (ignoring the sequencer limit bug). However, it is possible to use Connection Groups to get beyond this limit. I have successfully created a connection group with over 6000 active fonts. If I needed them I could probably get all 60,000 fonts that I originally downloaded working in a single connection group. Bring on the Mac users!

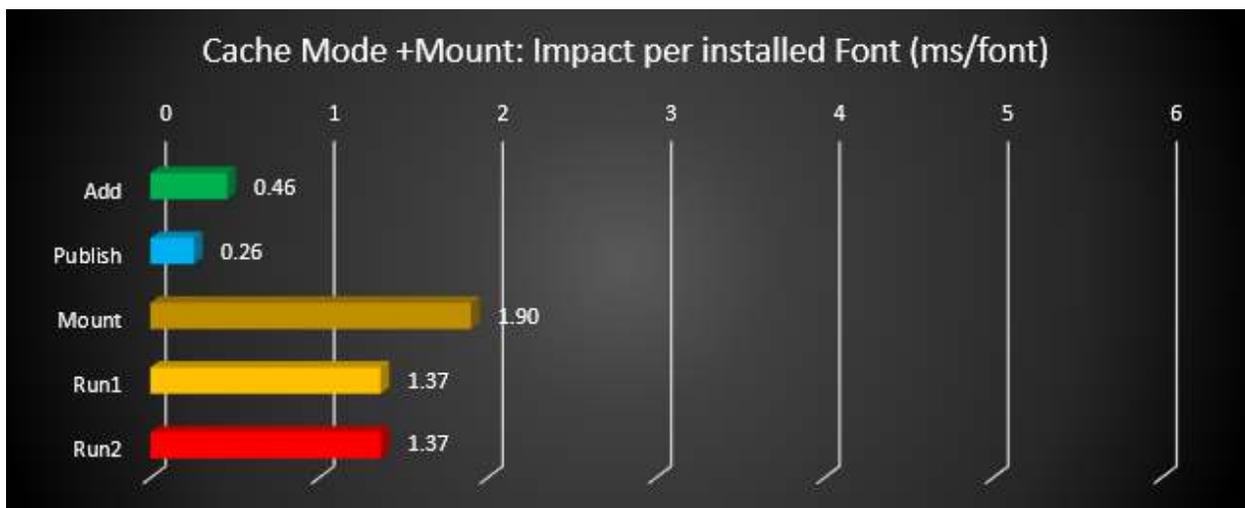
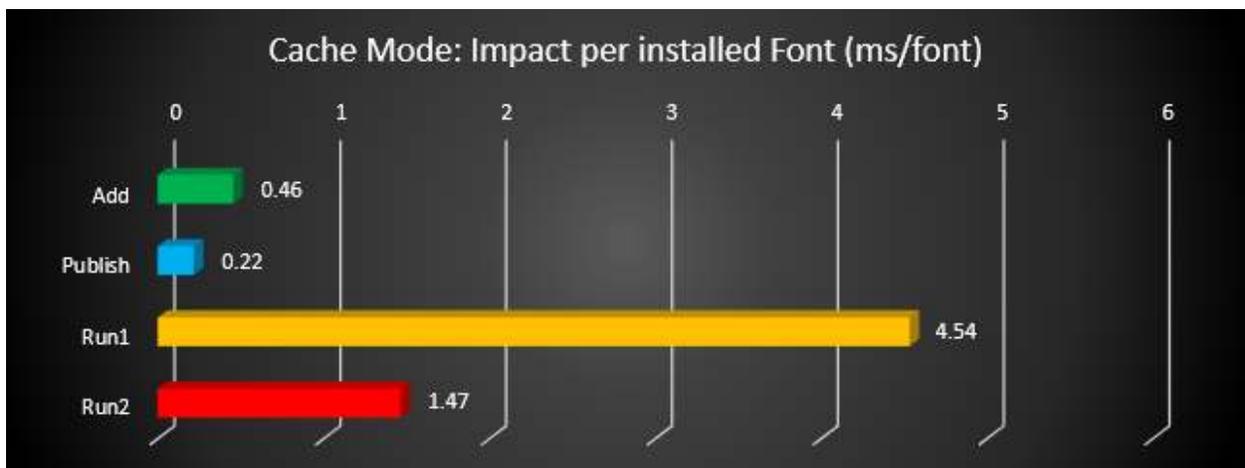
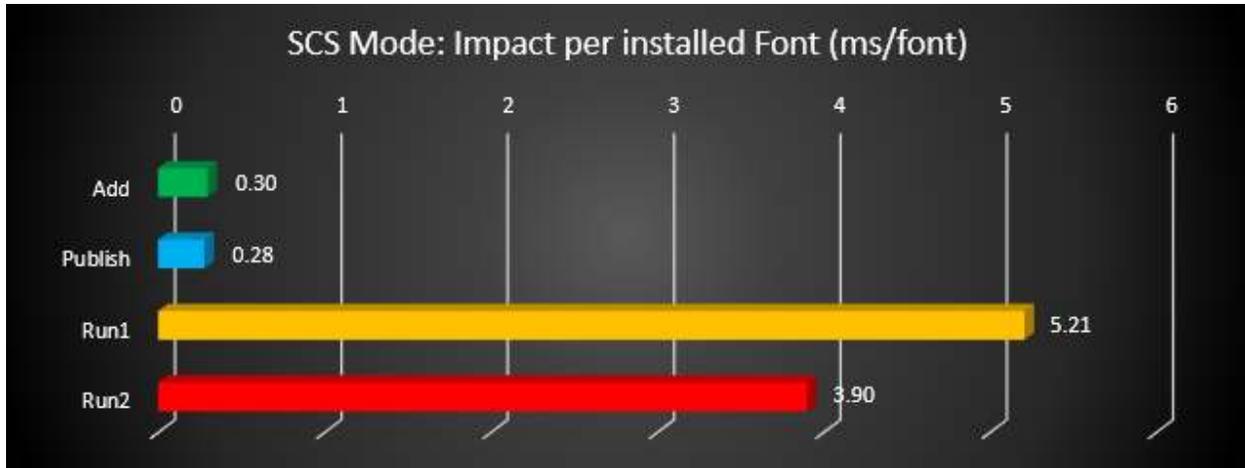


A SINGLE APP-V PACKAGE IS LIMITED TO AROUND 870 FONTS BY WINDOWS. PERFORMANCE ASIDE, THERE APPEARS TO BE NO LIMIT TO THE NUMBER OF ACTIVE FONTS IN A CONNECTION GROUP, ALLOWING US TO EXCEED THE NATIVE LIMIT OF ABOUT 1000 FONTS.

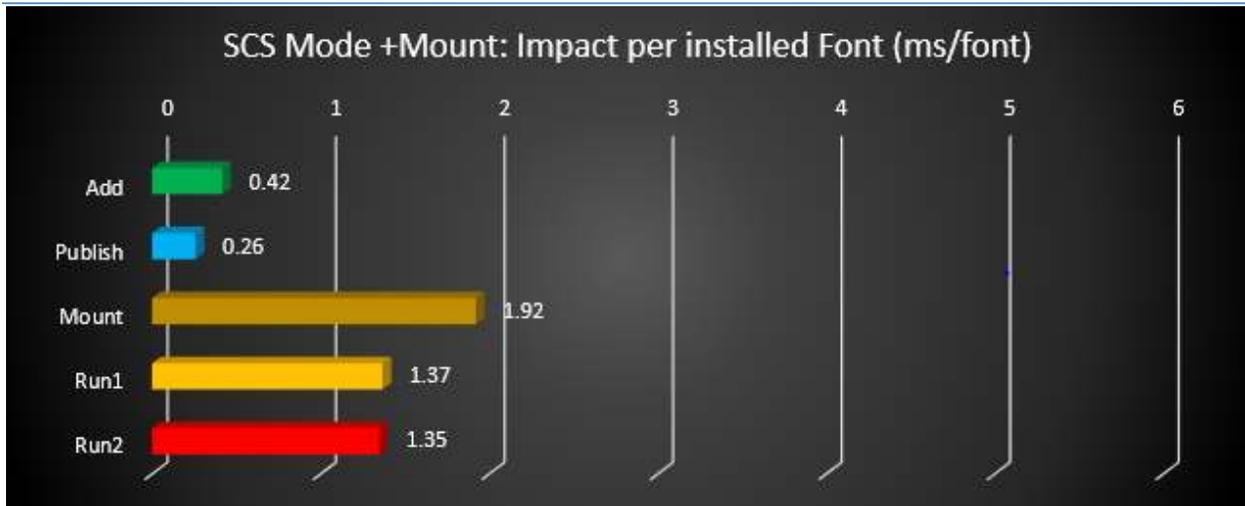
This leads to finding number 4:

4.3 Standard Scenario Testing results

The testing results, comparing the addition of installed fonts to a package to an identical package without the installed fonts. Three different common scenarios are shown.



For completeness, I added a fourth scenario, which might represent launch times when SCS mode is used but pre publish/caching is performed of the app using an imaging technology.



The calculations used to generate these graphs come from a package containing 770 fonts and assumes that the impact of each font is equal. In reality, the first font will have a greater impact as that triggers the font subsystem, but you can still use the calculated numbers to determine how much faster launch time will be if you remove all of the fonts from a package by multiplying the appropriate number times the number of fonts inside the package.

4.4 Results Interpretation

From these test results, and of the more detailed results presented in a later section of this paper, a number of conclusions may be made.

Fonts affect virtual application performance in three ways:

- Because fonts are also files, the font file adds overhead to the package. In addition to the increase in overall package size, the Central Directory is larger, and the FilesystemMetadata, and BlockMap files are bigger. This affects the time to complete the Add-Package operation somewhat.
- Additionally, First-Run is impacted when the reparse points and sparse block settings for the files are established. The impact of these files are felt whenever streaming of the font file is required. This may be seen in mount (or auto-load) operations, or potentially as part



A PACKAGE OF 500 FONTS WILL ADD 1/4 SECOND TO THE ADD-PACKAGE STEP.

of the publishing block is stream training is performed, or during first and every start of the virtual environment.

- Because installed fonts are also registry entries, an installed font increases the size of the virtual registry. This impact would occur as Add, Publish, and First Run steps, however the impact of this on performance is too small to be detectable.
- As an internal extension, fonts adds time to the startup of the virtual environment every time the virtual environment is started.

Because fonts are “internal extensions”, it seems that the addition of fonts to a package have little impact on the time to complete the Publish-Package step other than increased xml processing time.



FONTS IN A PACKAGE HAVE A SMALL IMPACT ON PUBLISH-APPVCLIENTPACKAGE.

While there is concern for Fonts performance impacts is during the logon deployment operations, it is during the launch of an application to start a virtual environment that include virtual fonts that has the greatest impact. While some of these impacts may not be specific to the number of fonts, but just the amount of stuff that requires processing at different points when any fonts are present, for the most part, the performance impact appears to be relative to the number of fonts.

The differences in start of virtual environment launch performance in the different scenarios is quite surprising. In particular, there appears to be a launch penalty that occurs when the package has not been 100% cached locally, even on secondary runs where everything needed should have been cached.



WHEN NOT PRE-CACHED, ABOUT 190 TO 220 FONTS IN A PACKAGE ADD 1 SECOND TO INITIAL RUN LAUNCH TIME.



WHEN PRE-CACHED, ABOUT 730 FONTS IN A PACKAGE ADD 1 SECOND TO LAUNCH TIME.

4.5 Improvement Options

The following options for improving performance of packages with fonts are identified:

1. Removing unnecessary fonts will improve performance. It would be rare that an application would add font files without installing them, but if this is detected they should probably be removed. More likely, it might be possible to remove installed fonts from the package. While requiring additional testing, this might be tried if an application adds a large number of fonts that are not needed. In most cases, the software will use a “close enough” font unless printing is involved.
2. Pre-caching will improve launch performance.
 - Causing the font files to be part of the publishing block would add to the publish-package step and improve launch somewhat, but not significantly as only 100% package caching seems to help significantly. Instead, full caching of the package should be considered.
 - This is easy to do using the “Force” option in the sequencer stream training phase.
 - As a Post-sequencing solution when the client is configured in caching mode, turning on Autoload=2 will help. Even the default setting of Autoload=1 will eventually help, after the first run.
 - As a Post-sequencing solution when the client is configured with or without SCS mode, scripting a mount command on the package will help. This is more likely to be used when pre-caching apps during imaging for use in SCS mode, but is equally applicable to RDS scenarios
3. Disabling the font-subsystem may also be considered. This may be done as a post-sequencing operation by editing the DeploymentConfiguration.XML file and only affects the one package. This change carries the same risks as uninstalling the fonts, but because the files and registry entries are still present it only improve launch time performance.

5 Testing Strategy Used

This section provides details about how the testing was performed.

5.1 About the Testing Platform

The testing results depicted in this paper are based on:

App-V 5.0 SP2 with HotFix 4 running on a Windows 7 SP1 x86 virtual machine.

The testing was performed in an isolated environment using a Microsoft 2012 R2 server with Hyper-V. The server has 24 processors and 64GB of RAM. To minimize external impacts, this server utilizes local storage and contains a VM with the domain controller. App-V Package sources were located on a share on this host.

The Test VM used had 2GB of RAM and was given 2 virtual CPUs. The App-V Client is configured for Shared Content Store mode (which disables background streaming and writing of fault-streamed data to the local cache).

5.2 About Test Packages and “Streaming Configuration”

All Test packages used are specially constructed software packages that I developed. These packages are generally stripped down to a bare minimum, except for an overabundance of the one particular things we want to measure when using this package. In many cases, this means custom software that I developed for the purpose of the test.

Unless specifically noted, each package was sequenced and configured for streaming by *not* launching anything during the streaming training configuration phase of the sequencer. This means that, barring mounting operations, almost everything in the package will fault-stream (stream on demand).

5.3 About the Testing Methods

All tests are automated using significant sleep periods before each portion of the testing to allow all systems to settle down, and warm-up of the external components (hypervisor/fileshare) and within the OS (App-V Client and drivers) are performed. The test process consists of:

- A **Test Cycle** that consists of a series of Test Passes.
- Each **Test Pass** consists of a number of Test Packages.
- Each **Tested Package** is tested using a series of actions and measurements.

A **Tested Package**, consists of a series of actions, always preceded by a significant sleep period to allow system background processes to settle down.

A *Test Pass* always starts from a freshly booted snapshot and with a dummy *Test Package* to warm up the App-V Client and Driver sub-systems. The results of this dummy package are not used.

A *Test Cycle* always starts with a *Test Pass* to warm up the external components of the Hypervisor and Windows File Share. Because the packages are relatively small compared to the amount of memory available, the packages are likely retained in memory in the Windows Standby Lists after the initial *Test Cycle*. These are described as follows, from the bottom up.

5.3.1 Test Package

For a given ***Test Package***, the series of actions includes:

- Waiting
- Add-AppVClientPackage
- Waiting
- Publish-AppVClientPackage
- Waiting
- [Optionally Mount-AppVClientPackage¹]
- Waiting
- First run (launch “cmd.exe² /c time /t” inside the virtual environment).
- Waiting
- Second run (launch “cmd.exe³ /c time /t” inside the virtual environment).

The time required for each of the actions to complete is recorded.

5.3.2 Test Pass

A ***Test Pass*** consists of testing multiple *Test Packages* as follows:

- Reverting the test VM to a snapshot.
- Waiting for the Hypervisor to settle.
- Booting the VM and logging in.
- Waiting.
- A series of actions and measurements on a warm-up package. These results are never used, it is only performed to warm up the client (client service, drivers, and WMI) and to ensure that each subsequent package fairly tested under similar conditions.

¹ With SCS enabled, mounting the package does result in the actual file content being stored in the App-V file cache. I test in SCS mode both with and without mounting to better delineate the cause of performance slowdowns on a package.

² This is used rather than a program in the package to produce a comparable time that varies based on special actions that the client must perform during virtual environment startup and shutdown due to the package content.

³ The client is also known to perform special actions the first time a virtual environment is used, so the second run is used for comparison to the first run.

- Waiting.
- A series of actions and measurements on the first package.
- Waiting.
- A series of actions and measurements on the second package.
- Etc...
- Recording results

5.3.3 Test Cycle

Finally, A **Test Cycle** consists of several consecutive test runs of the same *Test Pass*. The first pass is used to “warm up” external systems and achieve a relatively consistent amount of caching by the server. The results of this pass are not used, but the results of the remaining passes are averaged to produce results. A Test Cycle typically requires a full day to complete.

5.4 About the Test Results Accuracy

As careful as I attempt to be to eliminate variability in the results, there is a fair amount of variability in results between two passes.

Due to the nature of the background interruptions affecting the results, the impact on result accuracy is felt much more on measurements that are shorter in duration than those that are longer. With measurements that are sub-second, this can produce results that typically vary by as much as +/-10% from the average. Unfortunately, I cannot wash out this effect by making a package with an extreme number of fonts due to OS and App-V limitations already discussed.

Instead, I use an approach to test with a sufficient number of test cycles and select the minimum value seen on any of the tests. The more repetitions that are made, the better this minimum value represents the time it takes for App-V to complete the task without the effects of any extraneous background interference.

It should be noted that only TTF fonts were tested as part of this work. There is a small potential of other font types having different performance characteristics.

6 Test Packages Utilized

This section details the packages used in testing.

All packages used in this test, including the base package, contain a small font installer application with a couple of shortcuts. The application is only used on the sequencer for installation; its inclusion in the package simulates a base application and makes it easy to test to see if the fonts are visible in the virtual package. This application is installed into the PVAD folder in each case. This allows for the evaluation of font impacts when packages/results are compared.

To separate out the impact of fonts as files of a certain size, and impact of detection, some of the packages have the font files present without installing, and present under a different file extension (and without installing).

6.1 Lots_OfNothing (Warmup)

This is a minimal App-V Package.

In developing this package, I discovered that there is an issue with the App-V Client in that there appears to be some sort of undocumented minimal package requirements. If you create a package with no registry entries, no files, and no integrations, the Add-AppVClientPackage cmdlet will error out with error 700002.

Therefore this package consists of one HKLM registry key, one HKCU registry key, one text file in the PVAD folder, and one shortcut (to the text file). Package Statistics⁴:

Size of .AppV File (Compressed)	26,639
Size of Central Directory	722
Size of BlockMap (Compressed)	615
Size of AppxManifest (Compressed)	793
Size of Registry.Dat (Compressed)	25,731
Number of Entries + EmptyDirectories	8+0
Number of Fonts Detected	0

⁴ Package Statistics are provided by a tool called "AppV_Manage" developed by the author. "Number of Fonts Detected" indicates the number recorded by the sequencer as fonts in the XML files; in some cases they will not be effective at the client.

This package is primarily used as the first package in a Test Pass, to warm up the OS and App-V Client components and dependencies⁵.

6.2 Lots_OfFonts_RenamedPVAD

This package consists of the Font Installer/Viewer program, plus 770 font files copied into the PVAD folder and renamed to an unknown extension, so as not to be detected by the sequencer. These fonts are not installed into windows, just the files placed there and renamed.

The sequencer treats these as just files.

They add to the overall size of the AppV file and to the CentralDirectory and BlockMap files. They also add to the number of Entries in the AppV file.

Package Statistics:

Size of .AppV File	33,283,040
Size of Central Directory	92,021
Size of BlockMap (Compressed)	64,715
Size of AppxManifest (Compressed)	1,047
Size of Registry.Dat (Compressed)	17,694
Number of Entries + EmptyDirectories	787+2
Number of Fonts Detected	0

⁵ When conducting tests that use mounting, I found it necessary to warm up the system without mounting this package. It appears that mounting this package causes an additional any subsequent Add-AppVClientPackage commands to take an extra around an extra second to complete. This issue only seems to exist with this package, and mounting other packages does not affect subsequent Add cmdlets. The cause of this is unknown.

6.3 Lots_OfFonts_RenamedVFS

This package consists of the Font Installer/Viewer program, plus 770 font files copied into a VFS folder and renamed to an unknown extension, so as not to be detected by the sequencer. These fonts are not installed into windows, just the files placed there and renamed.

The sequencer detects these as just files. They add to the overall size of the AppV file and to the CentralDirectory and BlockMap files. They also add to the number of Entries in the AppV file.

Package Statistics:

Size of .AppV File	33,343,913
Size of Central Directory	117,660
Size of BlockMap (Compressed)	65,314
Size of AppxManifest (Compressed)	1,061
Size of Registry.Dat (Compressed)	20,535
Number of Entries + EmptyDirectories	789+2
Number of Fonts Detected	0

6.4 Lots_OfFonts_CopiedPVAD

This package consists of the Font Installer/Viewer program, plus 770 font files copied into the PVAD folder. These fonts are not installed into windows, just the files placed there.

The sequencer detects these as fonts, and includes them in the font list in the AppXManifest (and external DeploymentConfiguration) file, but they are not installed and are not seen as fonts by applications running in the virtual environment.

They add to the overall size of the AppV file and to the CentralDirectory, BlockMap, and AppxManifest files. They also add to the number of Entries in the AppV file.

Package Statistics:

Size of .AppV File	33,285,865
Size of Central Directory	89,170
Size of BlockMap (Compressed)	64,713
Size of AppxManifest (Compressed)	5,293
Size of Registry.Dat (Compressed)	16,036
Number of Entries + EmptyDirectories	789+2
Number of Fonts Detected	770

6.5 Lots_OfFonts_CopiedVFS

This package consists of the Font Installer/Viewer program, plus 770 font files copied into a VFS folder. These fonts are not installed into windows, just the files placed there.

The sequencer detects these as fonts, and includes them in the font list in the AppXManifest (and external DeploymentConfiguration) file, but they are not installed and are not seen as fonts by applications running in the virtual environment.

They add to the overall size of the AppV file and to the CentralDirectory, BlockMap, and AppXManifest files. They also add to the number of Entries in the AppV file.

Package Statistics:

Size of .AppV File	33,341,033
Size of Central Directory	115,350
Size of BlockMap (Compressed)	65,234
Size of AppxManifest (Compressed)	5,482
Size of Registry.Dat (Compressed)	18,040
Number of Entries + EmptyDirectories	789+2
Number of Fonts Detected	770

6.6 Lots_OfFonts_Installed

This package consists of the Font Installer/Viewer program, plus 770 font files installed into the Windows/Fonts folder (which is in the VFS area). The original source font files are not included in the package.

The sequencer detects these as fonts, and includes them in the font list in the AppXManifest (and external DeploymentConfiguration) file, and they are seen as fonts by applications running in the virtual environment.

They add to the overall size of the AppV file and to the CentralDirectory, BlockMap, AppXManifest, and Registry files. They also add to the number of Entries in the AppV file.

Package Statistics:

Size of .AppV File	33,518,711
Size of Central Directory	89,569
Size of BlockMap (Compressed)	65,484
Size of AppxManifest (Compressed)	5,198
Size of Registry.Dat (Compressed)	36,852
Number of Entries + EmptyDirectories	792+3
Number of Fonts Detected	770

7 Detail Test Results

This section provides additional details of the testing results not reported in the summary.

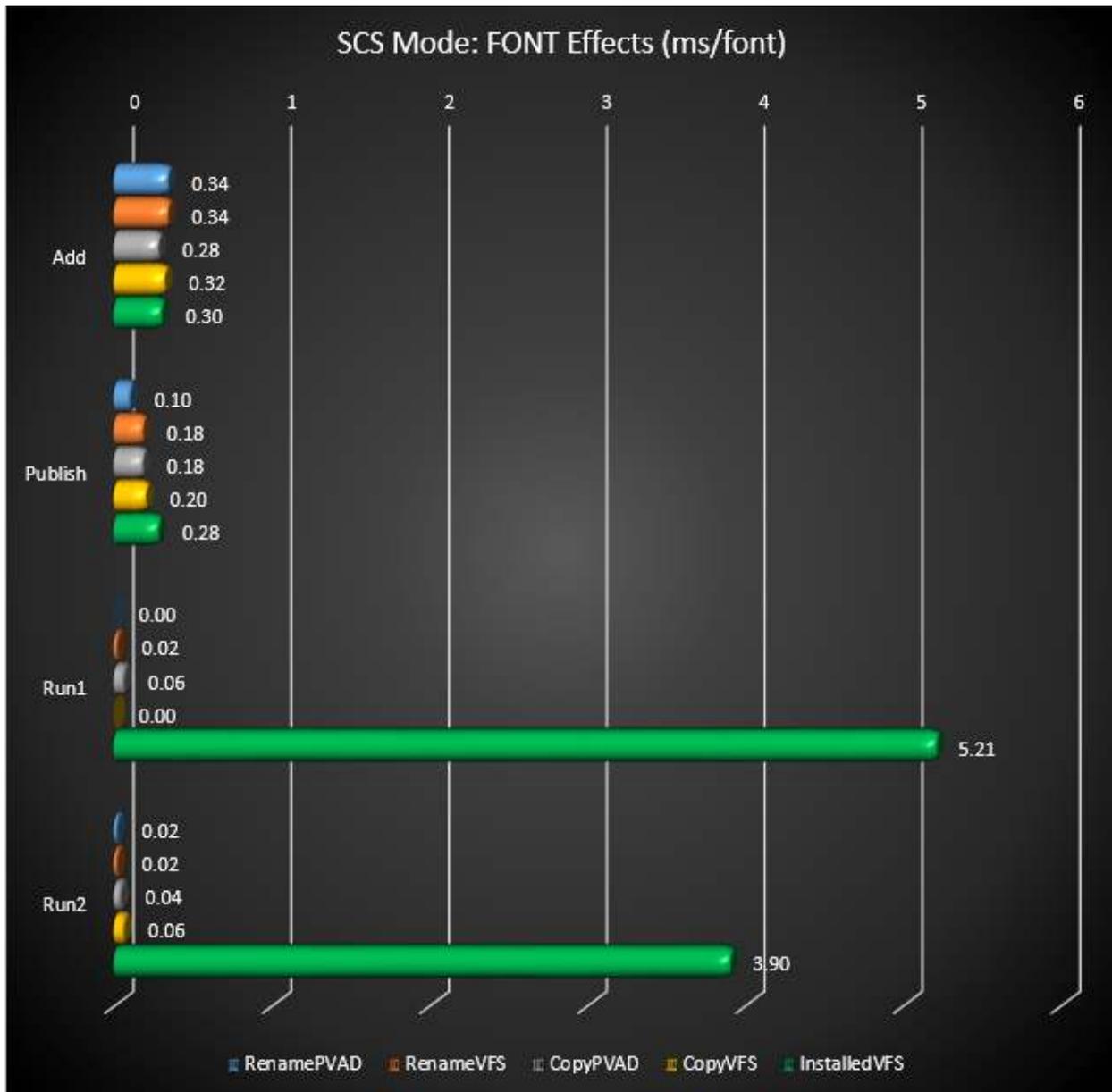
Results reported are based on an ideal test environment. Performance impacts identified in this paper will be very different in production environments. Specific numbers are *only* useful in comparison to numbers from other research papers in this series!

The simplified results provided earlier in the summary are based on substantially detailed testing, some of which the results are presented here. These tests are not necessarily scenario based, but are designed to illuminate the actions of different parts of the App-V client. These details allow for a more complete understanding of where deployment time impacts come from.

7.1 SCS Mode Without Mounting

In this test, Shared Content Store Mode is enabled as would typically be used. These results show the performance that you can expect when SCS Mode is enabled.

These results may also be interpreted to show the performance that you can expect without SCS mode enabled when background streaming is not enabled and font files are not included in the streaming configuration, except for the Second run.



From this (and procmon traces) we determine that the font subsystem reads in and processes each of the installed font files at the start of the virtual environment each time the environment is spun up. In an SCS mode, this means that the installed font files must be read over the network as part of this processing.

At the start of the virtual environment, and font subsystem, only those files detected by the sequencer and properly registered are required to be read in. Even when the sequencer detects the copied files as fonts and lists them for the font subsystem, without registration, the start of the virtual environment and startup of the font subsystem does not trigger them to be read in.

The slightly shorter time seen in the results for the second run for the case when fonts are installed is expected according to Microsoft. During the first run, the final registry staging to the user must occur so we expect the first run to take a little longer than the subsequent runs because the package virtual registry is larger when those fonts are registered. The cause of the additional overhead for the first run with registered fonts appears to be something the client does to prepare the virtual font subsystem for this package.

From the numbers we reach conclusion numbers 1 and 2:

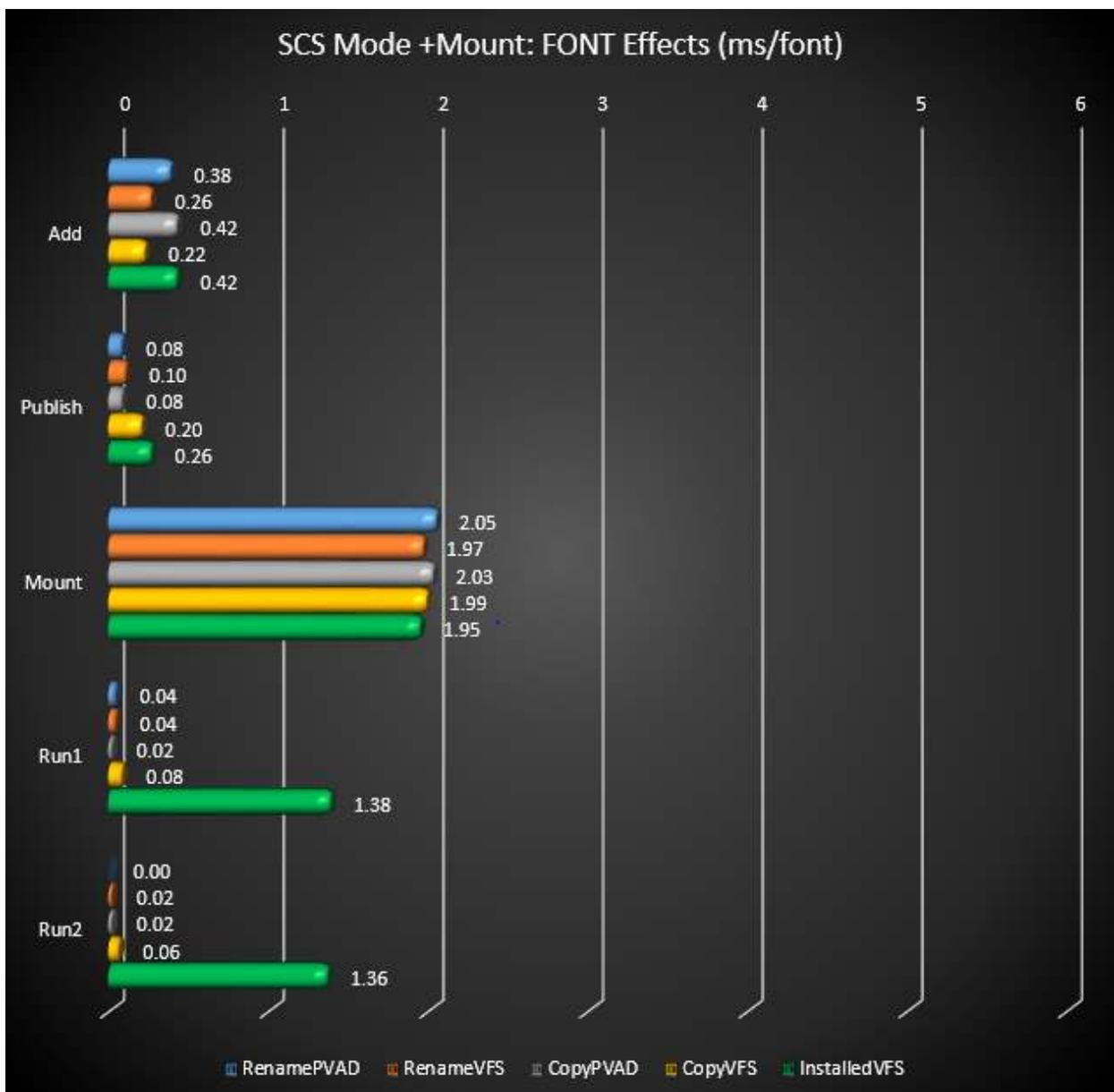


From this, and testing shown in the next section, we can recommend that for a package containing a large number of registered fonts, forcing the fonts to be locally cached, via mounting or forcing into the streaming training, can help with package launch times, saving about 3ms per font.

7.2 SCS Mode Testing with Mounting

In these tests, SCS Mode is enabled, however, a mount operation is performed prior to running the packages (which performs caching of the package locally, even when SCS mode is enabled). In SCS mode, the autoload setting is ignored (automatically disabled), but mounting may be performed. Although unusual, I wanted to test the possibility of pre-loading certain apps while in SCS mode – which could be done using imaging technology.

The chart which follows shows the results calculated from tests on the various font packages in these tests.



The times shown are calculated as the time for the given package minus the time for the BasePkg divided by the number of font files, producing a calculated impact in ms/font for each operation.

By comparing the first and second run results of this test against those of the SCS mode without mounting, we can determine how much of the font overhead was processing and how much was streaming the fonts down. However, those numbers do not add up! If you add the mount time to the first run time above you get about 3.3ms/font, but this compares to a 5.2ms/font measurement of the un-mounted SCS mode first launch.

It is possible that when the App-V file is purposely un-fragmented on the source media (which it was in this case), that the mount command may be more efficient at loading than I/O pattern generated by the font subsystem reading in all of those files. It is also possible that the mount command is simply more efficient than fault streaming.

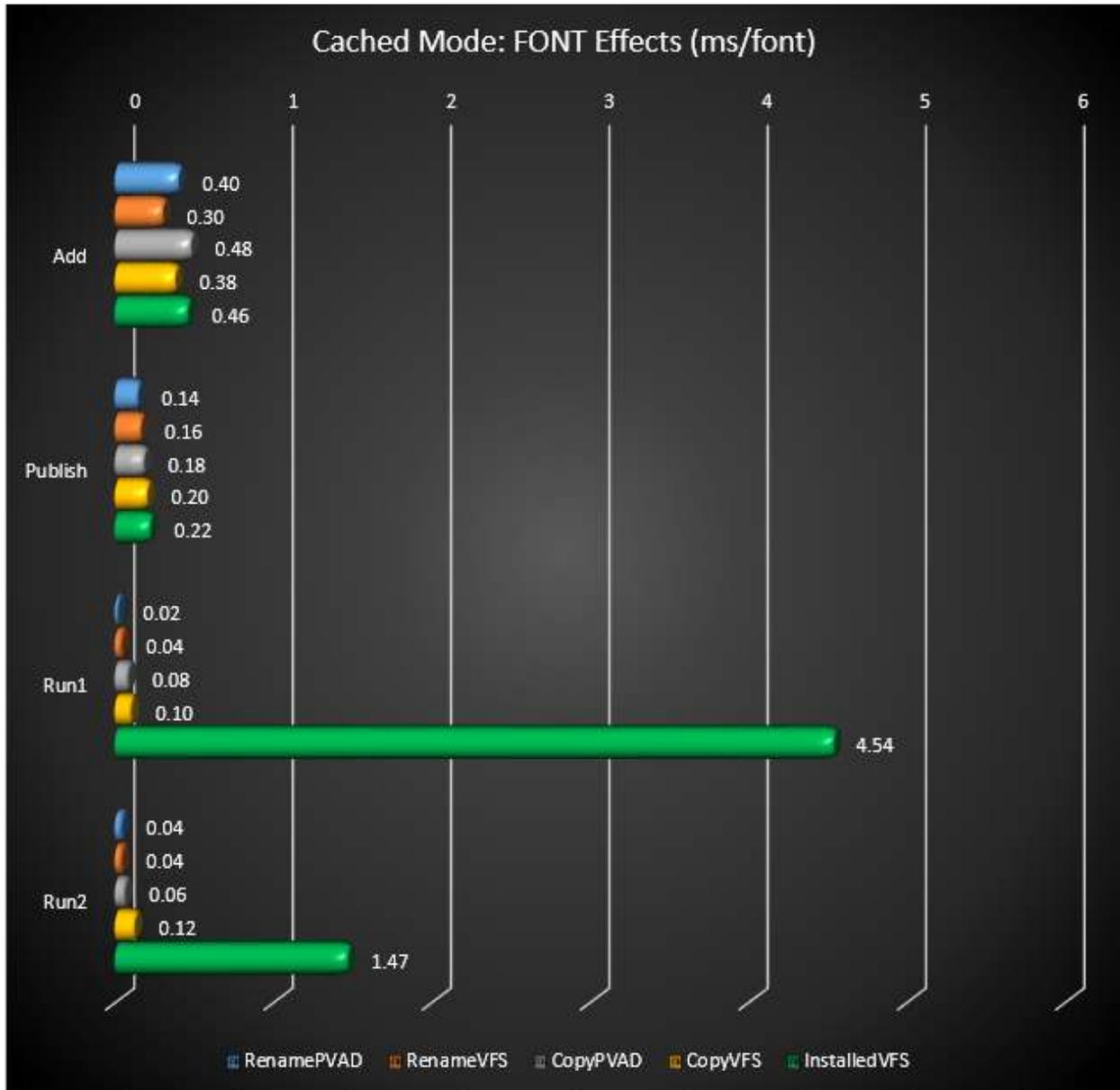
Conclusion number 3:



IN SCS MODE, EACH FONT ADDS ABOUT
1.4MS TO THE START OF THE VIRTUAL
ENVIRONMENT IF THE FONT FILE IS ALREADY
STREAMED LOCALLY.

7.3 SCS Mode Disabled, No Mounting

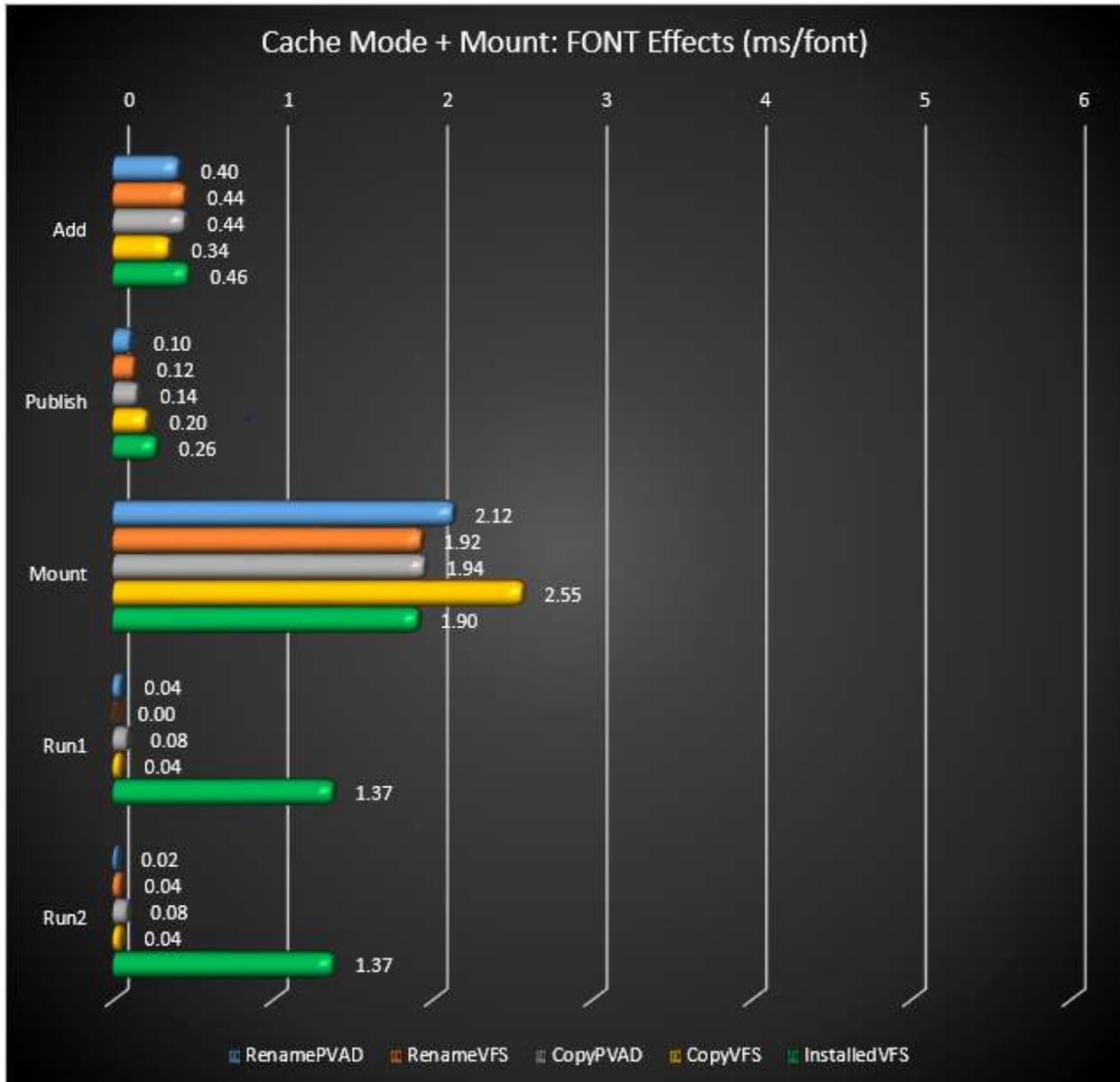
In these tests, SCS Mode is disabled and autoload is also disabled. This is closest to the default setup for clients (where autoload is enabled only for apps previously run) but I completely disabled autoload to keep things simpler.



For the package with the installed fonts, the 1st Run for this scenario is right in line with that of SCS mode. But the high amount of time to start the virtual environment for the 2nd Run in this series (when all of the font files should be locally cached) is surprising, especially when viewed against the results in the mounted case that follows. The cause of this is currently unknown.

7.4 SCS Mode Disabled, Mounting Used

In these tests, SCS Mode is disabled and autoload is disabled, but a Mount operation is performed after publishing. This test simulates autoload=2 setting when the user doesn't run the app for a significant period of time after publishing to allow all packages to get fully loaded, but allows for more accurate measurements.



NOTE: The mount time for CopyVFS should probably be about 2.0 ms/font. The result shown here may be due to an uncaught testing error.

8 About This Research Paper Series

This research paper is part of a series of papers, released by TMurgent Technologies, that investigate the performance impacts that certain application contents can have in the deployment of Microsoft App-V 5 packages.

Through these papers, we can better understand what areas to focus on when packaging applications for App-V when deployment and end-user experience is important. Additionally, with an understanding of these papers you can better target a specific package that is performing poorly and prioritize your efforts to improve it.

TMurgent Technologies, LLP is based in Canton, MA, USA; just 17 miles south of the offices where Microsoft develops the App-V product. TMurgent's Tim Mangan has a long history with the product, having built the original version at Softricity more than a dozen years ago. TMurgent is well known in the App-V community as a source for the best training classes on App-V as well as an endless supply of tools and information. More information is available at the website, www.tmurgent.com