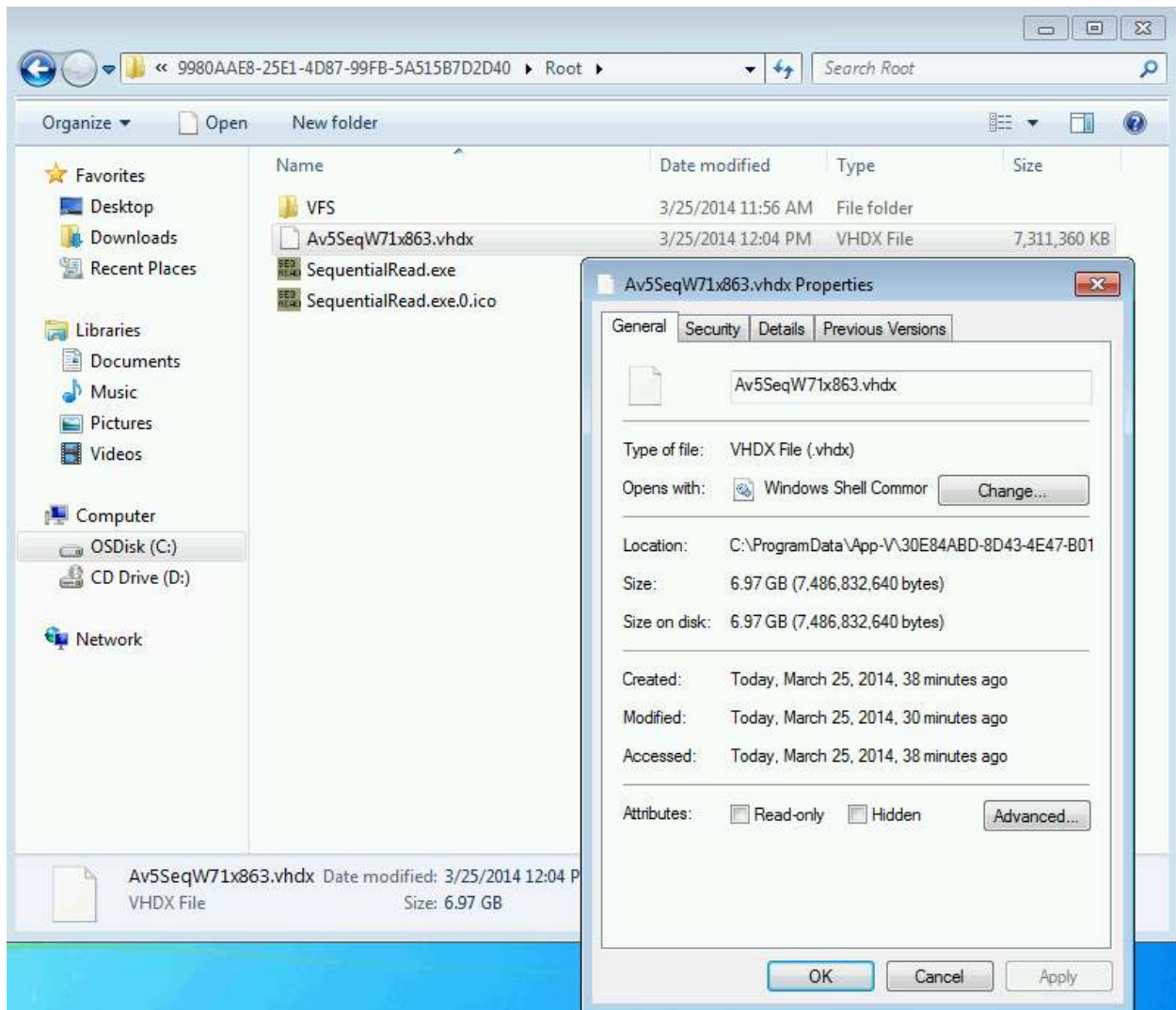


Effects of “A Really Big File” in App-V 5 SP2 Deployment Performance

TMurgent Performance Research Series



June, 2014

Contents

1	Introduction.....	3
2	Background on Really Big Files.....	4
2.1	Detecting Reparse Points and Sparse Blocks and how much is cached.....	4
3	Summary of Where Impacts of Files Are Felt.....	7
4	Testing Strategy Used.....	8
4.1	About the Testing Platform.....	8
4.2	About Test Packages and “Streaming Configuration”.....	8
4.3	About Special Testing for this Paper.....	8
4.4	About the Testing Methods.....	9
4.4.1	Test Package.....	9
4.4.2	Test Pass.....	10
4.4.3	Test Cycle.....	10
4.5	About the Test Results Accuracy.....	10
5	Test Packages Utilized.....	12
5.1	Lots_OfNothing (Baseline).....	12
5.2	Sequential Read with Big File.....	13
6	Detail Test Results.....	14
6.1	Standard Testing.....	14
6.2	Testing using the SequentialRead program inside the VE.....	15
6.2.1	Mounting versus Stream-Fault.....	16
6.2.2	SCS versus Caching.....	17
7	About This Research Paper Series.....	18

1 Introduction

The purpose of this research paper is to document the effects that a really large file has in Microsoft App-V Virtual Application Packages.

The effort is squarely aimed at answering questions on how such bulk data files in a package affect package 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.

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 Really Big Files

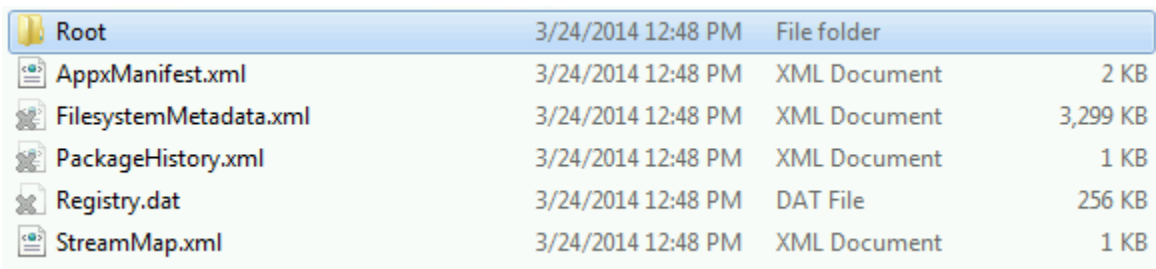
There is nothing special about really big files. They just take up lots of bandwidth and lots of disk space. But as we found out last year, with less than perfect software handling large files, they can also take up an awful lot of CPU to handle them.

This paper looks at issues discovered with large files last year against App-V 5.0 SP1 and updates the results using the currently latest software, which is so much better! This paper may also be useful in conjunction with other papers to understand the impacts of larger packages that you might choose to deploy over App-V.

2.1 Detecting Reparse Points and Sparse Blocks and how much is cached.

When the App-V Client brings in a package it initially creates placeholders for all of the files and gives each file record two special attributes, marking the file with a “Reparse Point” and a “Sparse” attribute. The Reparse Point “tags” the file so that the App-V Streaming driver knows that this is a file that might require streaming.

To determine if a package file is actually fully cached at the client, windows explorer provides a visual hint.



Root	3/24/2014 12:48 PM	File folder	
AppxManifest.xml	3/24/2014 12:48 PM	XML Document	2 KB
FilesystemMetadata.xml	3/24/2014 12:48 PM	XML Document	3,299 KB
PackageHistory.xml	3/24/2014 12:48 PM	XML Document	1 KB
Registry.dat	3/24/2014 12:48 PM	DAT File	256 KB
StreamMap.xml	3/24/2014 12:48 PM	XML Document	1 KB

In this image above, files with the grey “X” on them are not fully cached. You can also right-click one of the files and request the Properties dialog (as shown on the cover of this document). Comparing the size (which represents the logical size of the file) and size on-disk fields *approximates* the completeness of the caching of the file.

If you are really curious about completeness, you should view the sparse blocks directly. You can download a free tool from the TMurgent website called “Test_Directory” for this purpose. Point this tool at the App-V cache and it will show you the detail of what sparse blocks are present on the files and what parts of the file they represent.

└─ dictionaries	DICTIO~1	-	-	-	-	FA=8208(0x2010):	Directory, Not Con...
└─ en-US.aff		3.19 KB	3.19 KB	SparseOnDiskBlock...	FA=8736(0x2220):	Archive, Sparse, N...	
└─ en-US.dic		609.48 KB	192.00 KB				
└─ extensions	EXTENS~1	-	-	-	-	SparseOnDiskBlock_Offset: 0	Length: 131,072
└─ {972ce4c6-7e08...}	{972CE~1}	-	-	-	-	SparseOnDiskBlock_Offset: 589,824	Length: 034,287
└─ icon.png		2.13 KB	2.13 KB	SparseOnDiskBlock...	FA=8736(0x2220):	Archive, Sparse, N...	
└─ install.rdf		1.29 KB	1.29 KB	SparseOnDiskBlock...	FA=8736(0x2220):	Archive, Sparse, N...	

Adding the reparse point and sparse information to the file increases the on-disk size of the file ever so slightly. But when portions of the file, such as the case in the image above, are not yet streamed into the local App-V cache the on-disk size can be small.

As can be seen in the illustration above and the other on the cover of this document, this all means that the amount of space actually taken up on disk for a given file might be either less, or slightly more, than the actual file.

When the stream driver notices a read request of a file tagged with an App-V Reparse point, it checks the sparse blocks similarly to the tool above and determines if the portion of the file requested is locally cached or not. If it is not cached, it places the read request on hold and queues up a streaming request to acquire the content. When the content arrives it is immediately handed off to the requesting application. Then, unless Shared Content Store (SCS) Mode is enabled, it writes the new sparse block to the file.

With SCS Mode enabled, the files (normally) have only the Sparse Attribute but no SparseOnDiskBlock records will be present. With SCS mode enabled, however, it is possible to force cache the contents of a package using the Mount-AppVClientPackage cmdlet or from the optional Client Management Console app.

When streaming is occurring, especially with a Big File, there is not only much network traffic, and possibly disk write operations, but additional CPU expended to perform decompression. All data files in the App-V package are compressed and remain compressed during transmission. The client uses a combination of the App-V Archive format “Central Directory” and the StreamMap.xml file contained inside the package to know which compressed block to stream over for decompression.

The decompression technology used allows each compressed block to be independently decompressed, so the compressed blocks may be delivered in any order. The size of each decompressed block varies, as the compression technique uses 64KB of uncompressed data that is compressed into the block.

Network traces indicate that the client appears to be smart enough to not ask for the same compressed block when the application first asks for only part of the block and then shortly later asks for another portion of the same compressed KB block. Due to queuing techniques, possible SCS

Mode use, and timing of the requests by the application this process might not be as efficient as it could be all of the time.

3 Summary of Where Impacts of Files Are Felt

It is important to keep in mind that this research is only a large files. A separate research paper will address a large collection of smaller files, which impact performance in very different ways under virtualization.

Large Files within a package affect performance in several ways:

- An increase in size of the .AppV compression central directory index causes more data to be streamed during the Add-AppVClientPackage step. Each 64KB block (size before compression) adds an entry into the index; a small file probably makes no difference, but a large file is detectable. The client reads the entire central directory when it opens the App-V file so that it may maintain a complete table of these indexes.
- During the Publish step (when the large file is not part of the publishing block), a smaller impact is also detected. This is believed to again be due to the large compression directory of the .Appv file.
- During Mount operations, either command or background streaming, the file is streamed over the network and written to the cache. Obviously file size and compressibility will impact the time taken.
- Additional performance degradation at runtime was also detected. This was unusually large in the first run, but still detectable in subsequent runs, even when the large file is not used in running the virtual application. The source of this is unknown at this time.

The removal of large and unused files from packages will improve package performance.

Using the Package Mount cmdlet to stream large packages is now much more efficient than runtime streaming. In previous testing against 5.0 SP1, I found there to be little difference in mount versus stream-fault performance.

The performance issues with large files with Shared Content Store Mode enabled that were seen in 5.0 SP1 are also now gone. In previous tests, either mounting or reading the file from the SCS store caused massive CPU utilization by the streaming driver that remained long after the application was done; crippling the system. *The results of this test eliminate CPU usage as a reason to avoid SCS Mode.*

Because these tests are conducted on a hypervisor with no competing VMs, SCS mode will always exhibit results that show less performance than cached mode. But such results should be quite different in a production environment where the penalty of write IOPS caused by caching the content to the VM becomes important.

4 Testing Strategy Used

This section details how the testing was performed.

4.1 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).

4.2 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).

4.3 About Special Testing for this Paper

In addition to the testing methods described in the following section, which are used for each paper in this series, additional testing was performed.

The additional test involves running a specially written application to sequentially read the big file. This test was performed in the same four different scenarios used in the standard testing methods.

4.4 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.

4.4.1 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).

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.

- 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.

4.4.2 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.
- 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

4.4.3 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.

4.5 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

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.

longer. With measurements that are sub-second, this can produce results that typically vary by as much as +/-10% from the average.

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.

5 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).

5.1 Lots_OfNothing (Baseline)

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

This package is used to provide a baseline for general package processing.

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.

5.2 Sequential Read with Big File

This package consists of an updated version of the SequentialRead program used when writing my book *“Windows System Performance Through Caching”*, along with a very large file.

SequentialRead is a utility that will read in a named file. There are options to control the read buffer size, and to request reading in the forward, reverse, or random order. I only used a 4KB buffer and requested reading only in the forward direction for all tests. This produces a sequential read of the file.

SequentialRead is a standalone exe utility that does not use an installer, so when creating the package the exe is manually copied into the PVAD folder and a shortcut is added.

The big file contained in the package is a system vhd file that is around 7GB in size. This file compresses to around 3GB inside the package. The big file is also copied to the PVAD folder.

Package Statistics:

Size of .AppV File	3,228,749,442
Size of Central Directory	2,974
Size of BlockMap (Compressed)	4,054,911
Size of AppxManifest (Compressed)	821
Size of Registry.Dat (Compressed)	29,113
Number of Entries + EmptyDirectories	26+2
Number of Fonts Detected	0

6 Detail Test Results

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!

6.1 Standard Testing

For consistency, the standard testing used in other papers in this series was performed, however the results of that testing was not terribly interesting as the BigFile primarily only affects mount results.

The added Big file does have some impact on the package add and publish steps as well as when running a cmd prompt inside the virtual environment, however this effect was fairly consistent for all four scenarios produced results of about:

- 0.2ms per MB for the add step,
- .02ms per MB for publish step,
- .1ms per MB for the first run.
- Subsequent runs were barely noticeable in the results.

6.2 Testing using the SequentialRead program inside the VE

Tests were performed using the SequentialRead program to read in the big file inside the virtual environment. These tests were also performed with and without Mounting in both the normal caching setup and also with SCS enabled. A summary of the mount and first Run results are shown here.



In situations where deployment performance is crucial, such as VDI scenarios, these results show the effect that file content can have. While there may not be much you can do about it, these results are important to understand as file portions that must be streamed affect the performance of many of the other tests in this series of papers.

6.2.1 Mounting versus Stream-Fault

The performance improvement of pre-streaming the package by mounting is different than in



THESE ISOLATED TESTS SHOWED MOUNTING PERFORMANCE TO BE ABOUT 27 SECONDS PER UNCOMPRESSED GB, AND STREAM-FAULTING OVERHEAD TO BE ABOUT 65 SECONDS/GB.

prior releases. Previously, even in the caching scenario, the sum of the mount plus run time when mounting was about the same as running without caching. It appears that work was done by the development team to improve mount performance in the release

that did not lend itself to improving fault-streaming.

In our test, the “BigFile” would be contained in contiguous compressed blocks, and the mount appears to bring down the file by serially asking for each stream entry (which is a 64KB chunk that is compressed). When left to fault-

streaming, the app is requesting 4k, which will cause the appropriate 64k block to be streamed. Because of Windows file system cache read-ahead⁵, the next 4k may be requested before the compressed block received.



MOUNTING THE PACKAGE, OR SELECTING THE “FORCE” CHECKBOX IN THE SEQUENCER, IS THE MOST EFFICIENT WAY TO STREAM FILES TO THE LOCAL CACHE.

Thus the client must implement a queuing method of some kind to avoid re-requesting the same compressed block again. It appears from these tests that fault-streaming can be half as efficient as mounting.

This impact might be smaller or greater on a regular production like application package that consumes large file I/O from the package. The application I/O pattern of such a package would probably look more like random I/O than sequential. On one hand, this makes the ultimate remote file-system less efficient which should cause a greater penalty for fault-streaming, but on the other hand if the pattern avoids the queuing issue that I suspect occurs in the driver it would help. I have not come up with a test scenario to reliably test this out.

⁵ See “*Windows System Performance Through Caching*” for an explanation.

6.2.2 SCS versus Caching

Although these results show Shared Content Store mode requiring about 3 seconds per GB more



THE IMPACT OF SCS IN PRODUCTION
ENVIRONMENTS CANNOT BE DEDUCED FROM
THESE TEST RESULTS.

time for either mount/run or run-without-mount scenarios, I do not believe that this translates into reduced performance in production scenarios.

In addition to reduced total disk storage requirements, SCS mode is needed to

reduce the impact of Write IOPS when a user is on a VM using centralized storage. But these tests were designed to eliminate competing write traffic. Testing in a multi-VM production style environment, such as would be done using LogIn VSI, would produce very different results. Depending on the implementation of the centralized storage, user performance might improve by enabling SCS mode. I look forward to my friends as PQR and Login Consultants publishing such test results.

7 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