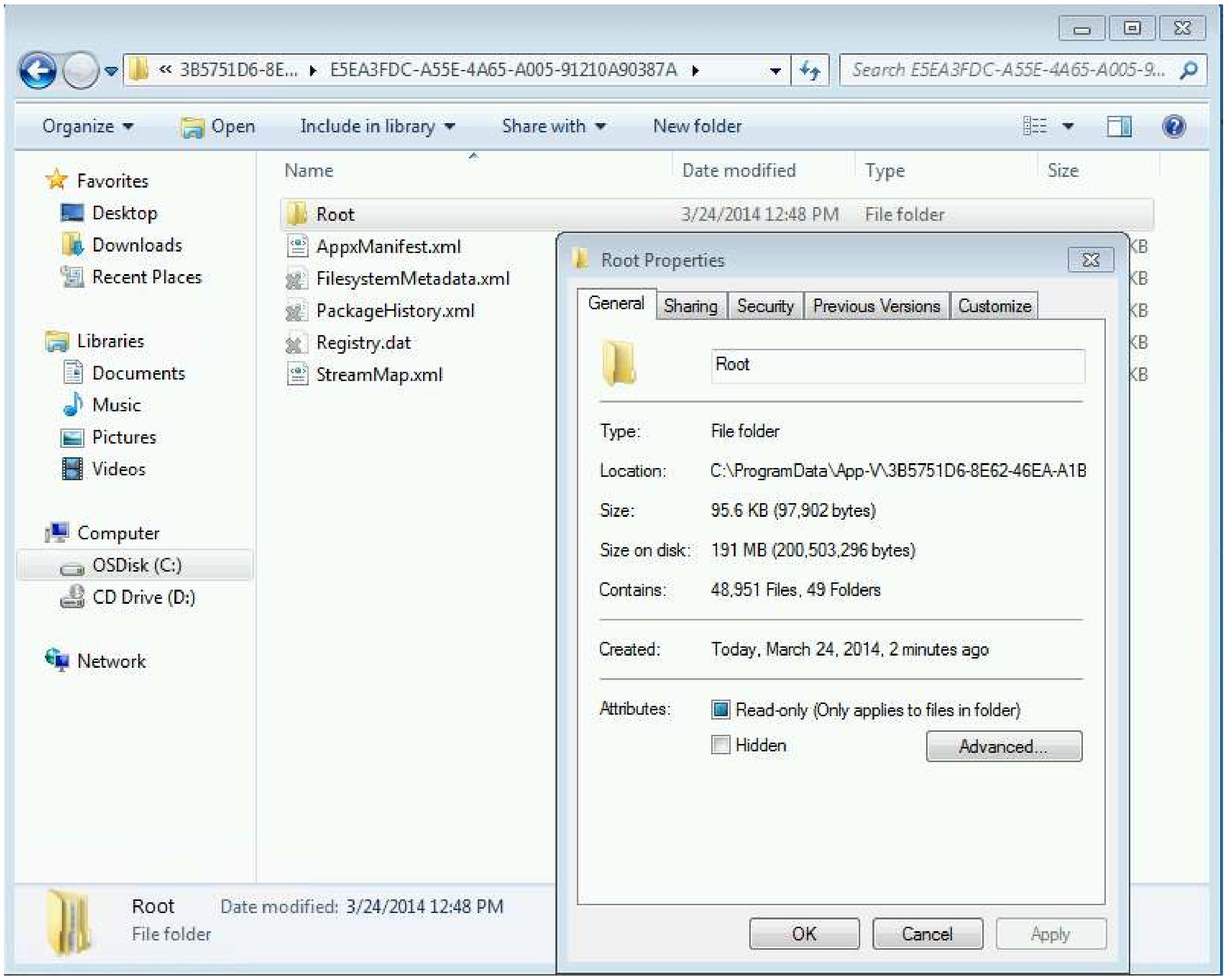
Effects of “Files” and “Folders” in App-V 5 SP2 Deployment Performance

TMurgent Performance Research Series



Name	Date modified	Type	Size
Root	3/24/2014 12:48 PM	File folder	
AppxManifest.xml			KB
FilesystemMetadata.xml			KB
PackageHistory.xml			KB
Registry.dat			KB
StreamMap.xml			KB

Root Properties

	Root
Type:	File folder
Location:	C:\ProgramData\App-V\3B5751D6-8E62-46EA-A1B
Size:	95.6 KB (97,902 bytes)
Size on disk:	191 MB (200,503,296 bytes)
Contains:	48,951 Files, 49 Folders
Created:	Today, March 24, 2014, 2 minutes ago
Attributes:	Read-only (Only applies to files in folder)
	Hidden

Root File folder Date modified: 3/24/2014 12:48 PM

June, 2014

Contents

1 Introduction

The purpose of this research paper is to document the effects that unused files and folders have in Microsoft App-V Virtual Application Packages. In this paper, the impact of a large number of folders and/or small files is examined. A separate paper will examine the impact of a large file.

The effort is squarely aimed at answering questions on how the deployment of packages with a large number of files 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.

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 the App-V File System

The App-V file system has two major parts that receive different treatment, those items that were captured on the sequencer under the designated Primary Virtual Application Directory (PVAD), and those that were captured elsewhere on the partition holding the Windows folder. The latter are referred to as VFSd files (for Virtual File System) as they are placed inside the App-V file under a folder called VFS. The primary difference in treatment of these two areas is that the client can assume complete isolation for PVAD files and folders, but must virtually integrate VFS files and folders with the regular client operating system's file system.

Items in the PVAD area are generally accessed by the client/application using simple redirection – the PVAD folder is treated as a unique location on the client that does not need to overlay on top of existing folders and files. Meanwhile items in the VFS must be overlaid by one of the App-V drivers (based on pellucidity settings) on top of existing, not-virtualized, folders and files.

But PVAD versus VFS is not the only difference to consider. There are also differences in treatment by the App-V client depending if the object is a file or a folder, and whether the folder is empty or not.

From a performance perspective, user changes to files also play a big part. Ultimately, when an application tries to locate a file not under the PVAD, the App-V client potentially has to look in up to three different places for the file; the user Copy-On-Write cache, the package VFS area, and the native file system. Additionally, the package content may be in memory, on the local file system cache, or currently residing in compressed form on a remote share. Placed in this light, it is amazing that we can get reasonable performance out of the system!

A particular product feature to consider, when discussing performance of the App-V File System is how the items are cached on the local system. A major feature of the client cache is called Shared Content Store (SCS) Mode, where only placeholders for the folders and files exist in the local cache and actual file content is streamed into memory from the package source on demand.

2.1 App-V Cache skeletal framework

Whether or not SCS mode is in use, the client must create placeholders on the NFTS partition that forms the App-V file cache for the package. These can be seen under the C:\ProgramData\App-V folder, by default. Here, you will find folders for each package, using the name of the PackageID GUID. Under that is a folder for the PackageVersion GUID, and under that is where the internals of the App-V file are exposed, through these placeholders.

The placeholders for each file and folder are real file system objects, but with some exotic properties added to them. The special properties used in particular by App-V are called reparse points and sparse blocks.

2.2 The Reparse Point

A Reparse Point is a file system property that acts as a signal to a file system filter driver. When the reparse point is set, a filter driver can query the file object to see if it is an object that is has to handle in a special way. All of the App-V package cache folders and files are marked with a reparse point by the App-V Client on creation to signal the App-V filter drivers, specifically the AppVStrm driver, to handle access to them specially.

2.3 The new "Sparse File"

Folders do not use the Sparse attributes in App-V. The folder definition, including child members fully exist in the on-disk image locally. Sparse, however, is used for files in both the PVAD and VFS.

Sparse Files were originally created as a general solution for saving space on a file system, but in particular was created for use by databases. Typically, the database wants to treat the database storage a large contiguous space, parts of which are paged in and out of memory. As things change in the database, it doesn't want to deal with the managing the file growing and shrinking operations, so it creates a fixed size file and then zeroes out portions not in use. To save on storage, designers created the idea of a sparse block, where zeroed out portions could detected and removed, with markers placed at the file system level to indicate what parts of the file were actually present on disk.

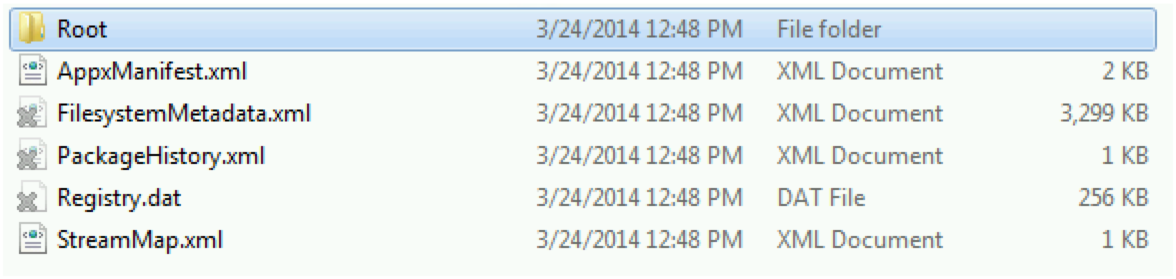
A Sparse File on disk consists of a set of sparse blocks and an overall length. Each sparse block has a starting offset and length. For a traditional Sparse File, any file part not contained in a defined sparse block is considered to be all zeroes. If you look at the properties of a sparse file, you can typically see that the two fields "file length" (overall length of the file) and "file space on disk" (size of the actual sparse blocks) are different.

App-V (SoftGrid originally) extended the concept of Sparse Block for streaming purposes. Instead of assuming that non-present sparse blocks are all zeroes, App-V uses this to indicate that those portions of the file need to be streamed. When a process tries to open and read one of these files, the AppVStrm driver first detects the reparse point, indicating that the file I/O operation is one that it should take action on. Once detected, the driver checks the list of sparse blocks currently present. When the read request is for a portion not currently cached on disk (or already in memory), it puts the read request on hold while it attempts to stream that portion of the App-V package from the source.

Assuming that the “local disk” is actually a virtualized disk file on a remote system, and that the streaming source has similar latency, then the amount of time to satisfy the request should be similar whether it is present in the “local” cache.

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

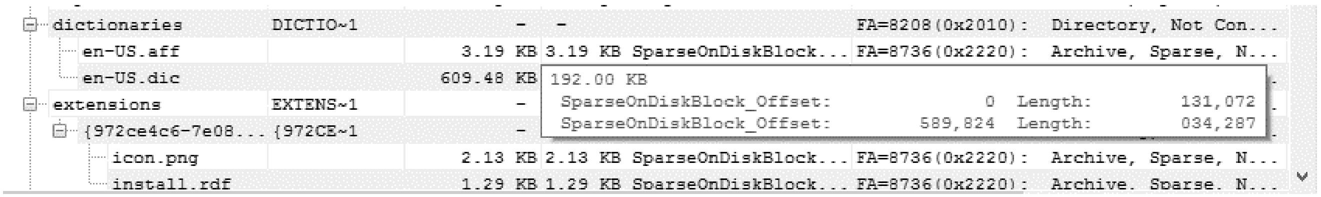
To determine if a package file is actually cached at the client, windows explorer provides a visual hint.



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 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
                                              SparseOnDiskBlock_Offset:           0  Length:        131,072
extensions            EXTENS~1          -     SparseOnDiskBlock_Offset:     589,824  Length:        034,287
  {972ce4c6-7e08... {972CE~1            -
    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...
```

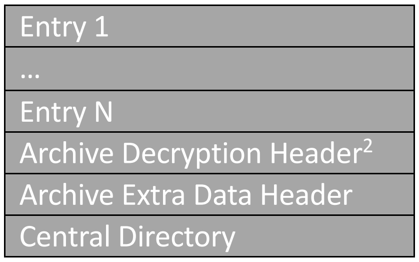
Adding the reparse point and sparse information to the file increases the on-disk size of the file slightly. But when portions of the file, such as the case in the image above, are not yet streamed into the local App-V cache the on-disk size can be small.

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 more, than the actual file.

2.5 Anatomy of the compressed .AppV file.

For the client to deploy an AppV file, even with SCS Mode enabled, it has to read in some significant portions of it. Understanding the format can be helpful in understanding the performance impact of certain packages. Microsoft App-V 5 packages are stored in a format that is based off the traditional “zip” compression file formats[1].

At a high level, the format consists of a number of entries, some optional headers, and a “Central Directory”, which is oddly placed at the end of the file. This is shown in the image on the right.



Microsoft App-V 5 stores the package contents in a zip compression based format that is specially controlled to make the complete package definition through the inclusion of additional information.

Compression files in general are constructed starting with a series of entry records representing a folder or file object (consisting of a local file header, encryption header, file data, and a data descriptor), some optional headers, and the Central Directory at the end of the file.

This format makes it easy to add a file by overwriting the end of records marker and directory map with the new file record, then rewriting the archive headers and updated directory (the remainder of the file need not be touched). Not that App-V works with the file this way, but that seemed to be the intent of the original zip format designers. It makes parsing the file unusual, as typically headers with locators are placed at the front of the file.

The App-V usage utilizes the relatively common “decode” compression algorithm which is supported directly by kernel interfaces. It also uses the Zip64 extensions, which has nothing to do with x86/x64 processors but allows the file package to exceed 4GB.

Another unique feature to the App-V format is that although directories may be specified as entries in the typical zip file, Microsoft does not. The existence of directory folders is instead inferred by the paths of file entries, and any folder that does not have a downstream file is not included in an entry at all. Instead, these are listed inside an XML file that the client parses to know about those folders[3].

[1] A generic format definition for the zip format can found from the originators of the ZIP format at https://www.pkware.com/documents/casestudies/APPNOTE.txt

[2] Optional; these two headers only used when Central Directory is encrypted. App-V does not encrypt the central directory.

[3] See Section 2.5.3 for details.

2.5.1 Central Directory Map

The Central Directory Map consists of a series of records, exactly one for each of the entries. The format for Zip was complicated at some point by the need to support larger archive files, called zip64 extensions. Originally, fields to hold items like file offsets and sizes limited the overall archive size to 4GB. The designers at PKware extended the format in a way to be able to write code that can read both old and new files. They chose to add optional indicators to signal that there are additional 8-byte fields containing these offsets and sizes. These fields are placed in "extra data" fields as part of central directory items and entries in the main archive. App-V always uses the zip64 extensions.

Central Directory Item 1
…
Central Directory Item N
Zip64 end of central directory record
Zip64 end of central directory locator
End of central directory record

Each of these central directory items includes the relative file name (path and file name relative to the start of the archive), compressed and uncompressed sizes, timestamps, CRC, file attributes, extra data, and on offset within the file pointing to the start of the entry. For any software to work with the compressed archive, it first needs to read the Central Directory.

When the App-V client first opens an appv file, it will typically read the first entry header signature at the front of the file (to make sure it looks like a zip based file). Then, it will read from the end of the file backwards until it can recognize the unique signature for the end of the central directory record. It then finds and reads the zip64 record and locator. Using the offset located there, it then reads in the central directory items (from first to last). By reading in this map, it knows where every file and folder record entry may be found as an offset in the file.

A Central Directory record for App-V takes 72 bytes, plus the length of the relative file path. As more files are added to the AppV file, the size of this central directory map grows. This affects App-V performance by adding to the amount of the AppV file that must be streamed over during package add.

Although not shown in this testing, the larger central directory map for the package also affects memory use by the App-V client, which will want to keep the directory block cached in memory (at least until the package is fully streamed to local cache). In SCS mode without local caching, this directory block would need to be read from the remote share on first reference to the package after each boot (whether by App-V Server Publishing or use).

2.5.2 Use of Entries in the Zip based format

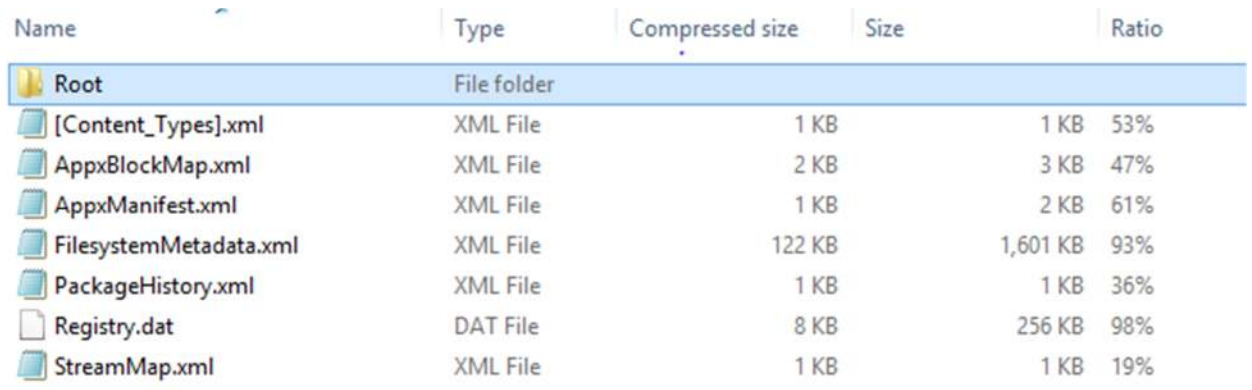
Local file header
Optional encryption header
File data
Data descriptor

An entry consists of a header, optionally an encryption header (not used in App-V), the (compressed) file data, and a data descriptor record.

The file header includes some of the information needed by NTFS, such as timestamps and CRCs, compressed and uncompressed sizes, relative file name, and a variable length “extra field”. Much of this is duplicated in the Central Directory.

Although the zip format allows vendor extensions to add additional information about the entry in the data descriptor, Microsoft seems to have chosen a different way to convey the additional information it must provide the client, such as Pellucidity and deletion markers. This information is conveyed in XML files that are part of the package.

The App-V package always contains a minimum set of seven files, six xml files and the registry.dat file.



Name	Type	Compressed size	Size	Ratio
Root	File folder			
[Content_Types].xml	XML File	1 KB	1 KB	53%
AppxBlockMap.xml	XML File	2 KB	3 KB	47%
AppxManifest.xml	XML File	1 KB	2 KB	61%
FilesystemMetadata.xml	XML File	122 KB	1,601 KB	93%
PackageHistory.xml	XML File	1 KB	1 KB	36%
Registry.dat	DAT File	8 KB	256 KB	98%
StreamMap.xml	XML File	1 KB	1 KB	19%

NOTE: The AppxBlockMap.xml file is the only of these files that does not get placed in the local App-V package cache. The only way to see this file is to look in the archive.

These xml files are integral parts of the package definition. Two of these files, AppxBlockMap and FilesystemMedata, contain information that is specific to the layout of the entries of the App-V file, which is another reason why you can’t edit a package with a standard zip utility without breaking the package.

But, frankly, the dependence on XML to convey all of this metadata is highly related to deployment performance in App-V 5. In the previous generation App-V system, this streamed metadata was encapsulated in the SFT file in binary form. XML requires considerable larger transfers to convey the same information, and (more importantly) it usually requires much more time to parse out XML data. Two of these files in particular, FilesystemMetadata.xml and AppxBlockMap.xml, have a tremendous impact on certain packages[4].

2.5.3 FilesystemMetadata.xml and the Special Effect of Empty Folders in App-V

Inside the App-V package is a special file named FileSystemMetadata.xml. This file contains all of the information for the client deployment to pre-stage reparse points and an empty sparse map for folders and files that will be "pre-staged" in the client cache when the package is added.

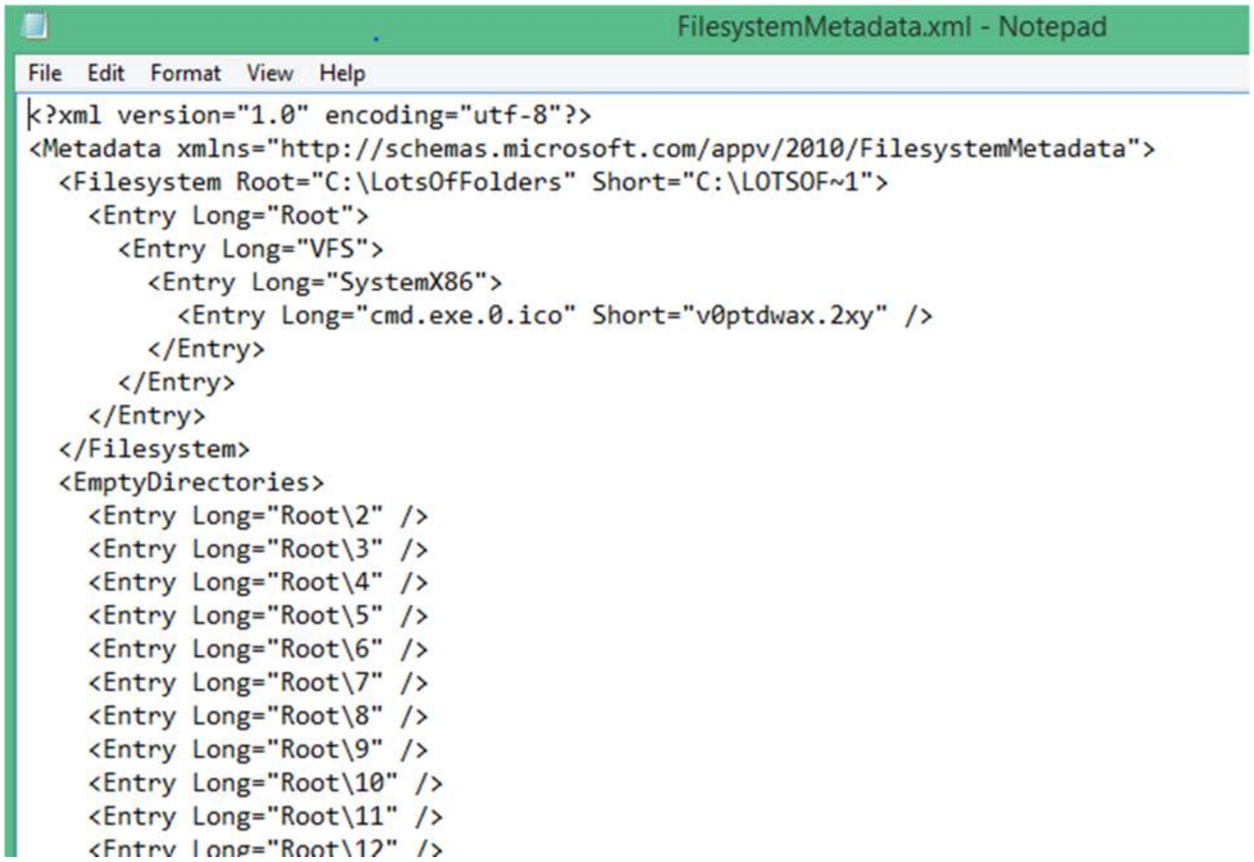
The FilesystemMetadata file contains three major elements:

- Filesystem: a list of every folder and file referenced as entries in the appv archive. This list includes both the files and referenced parent folders as separate entries.
- EmptyDirectories: a list of all additional folders that is in the package but not included in the archive because it had nothing other than other empty folders below it.
- OpaqueDirectories: a list of all folders that were marked "merge with local".

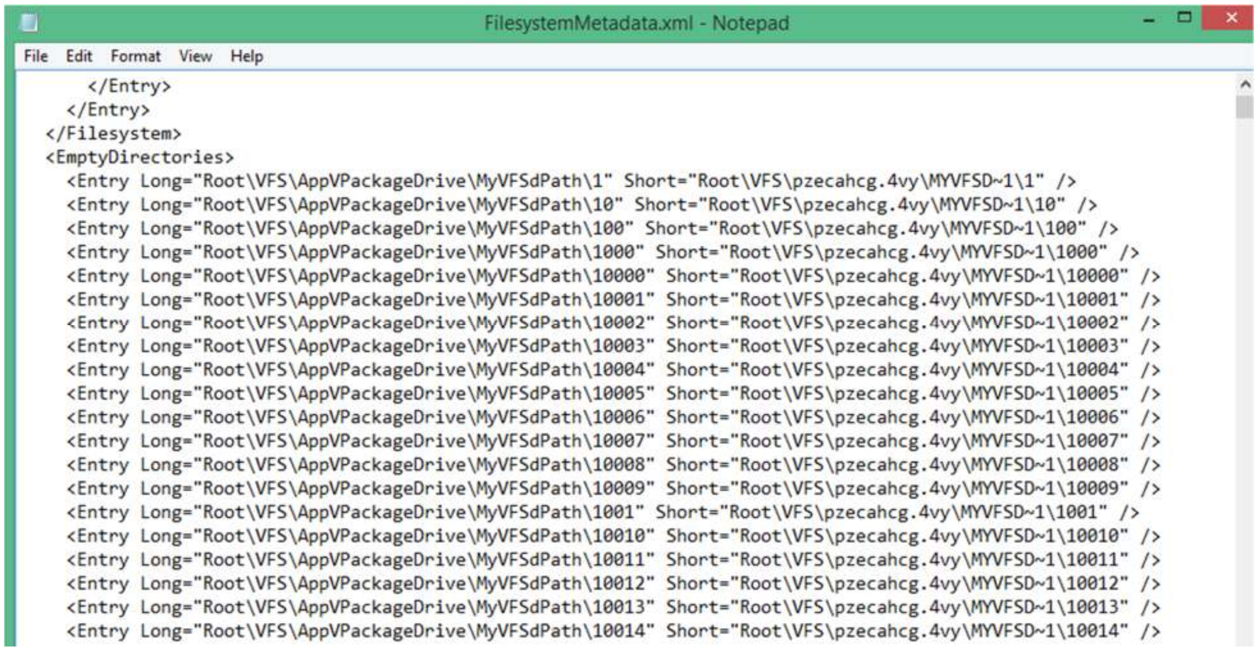
The images that follow (next page) show entries in this XML file for two of the packages used in the testing. The first had empty folders in the PVAD area, and the second in the VFS area[5].

[4] StreamMap.xml can also have deployment impact when stream training is configured during sequencing.

[5] You might also notice how for the EnptyDirectories the VFS folders (and files) have short-names recorded while PVAD ones do not. Oddly, both VFS and PVAD files listed in the Filesystem have short names. This design choice may adversely affect certain integration situations, but is outside the scope of this research. I'm sure Nicke will run into it.


FilesystemMetadata.xml - Notepad
File Edit Format View Help

```
<?xml version="1.0" encoding="utf-8"?>
<Metadata xmlns="http://schemas.microsoft.com/appv/2010/FilesystemMetadata">
  <Filesystem Root="C:\LotsOfFolders" Short="C:\LOTSOF~1">
    <Entry Long="Root">
      <Entry Long="VFS">
        <Entry Long="SystemX86">
          <Entry Long="cmd.exe.0.ico" Short="v0ptdwax.2xy" />
        </Entry>
      </Entry>
    </Entry>
  </Filesystem>
  <EmptyDirectories>
    <Entry Long="Root\2" />
    <Entry Long="Root\3" />
    <Entry Long="Root\4" />
    <Entry Long="Root\5" />
    <Entry Long="Root\6" />
    <Entry Long="Root\7" />
    <Entry Long="Root\8" />
    <Entry Long="Root\9" />
    <Entry Long="Root\10" />
    <Entry Long="Root\11" />
    <Entry Long="Root\12" />
```


FilesystemMetadata.xml - Notepad
File Edit Format View Help

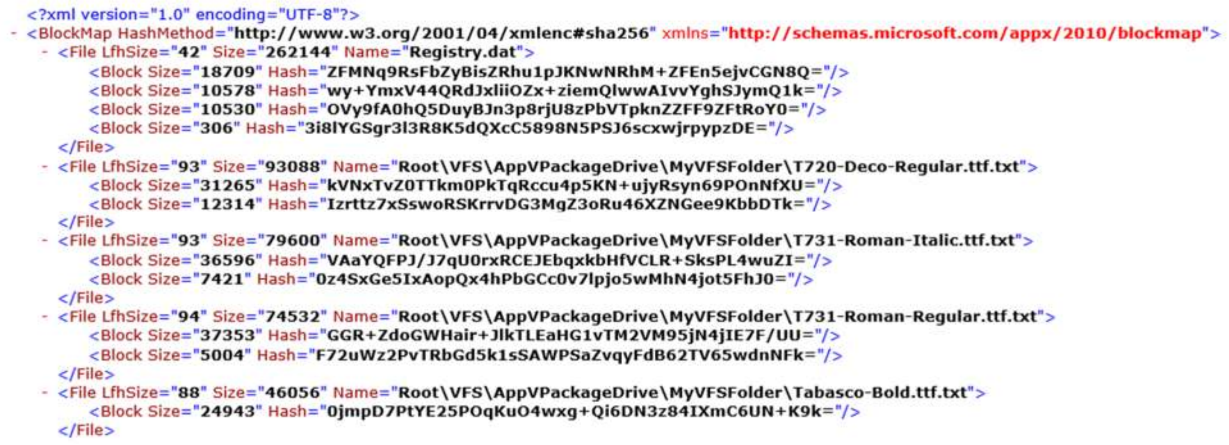
```
      </Entry>
    </Entry>
  </Filesystem>
  <EmptyDirectories>
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\1" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\1" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\100" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\100" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\1000" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\1000" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10000" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10000" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10001" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10001" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10002" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10002" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10003" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10003" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10004" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10004" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10005" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10005" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10006" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10006" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10007" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10007" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10008" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10008" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10009" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10009" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\1001" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\1001" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10010" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10010" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10011" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10011" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10012" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10012" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10013" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10013" />
    <Entry Long="Root\VFS\AppVPackageDrive\MyVFSdPath\10014" Short="Root\VFS\pzecahcg.4vy\MYVFSD~1\10014" />
```

This is important to note as the size of this XML file affects processing performance during deployment as the file is completely read in during the add phase.

2.5.4 AppXBlockMap and Performance

The AppXBlockMap contains information for each file with data in an entry.

When a file is deflated (the compression technique used in App-V), the deflation works on 64k blocks of the original file. When decompressing, each block is individually decompressed. This makes it possible to stream and decompress only those portions of the file needed at the time, rather than decompressing the entire file or even entire archive. For each of the files with data, the AppXBlockMap records the compressed size of each of the blocks.



```
  <?xml version="1.0" encoding="UTF-8"?>
- <BlockMap HashMethod="http://www.w3.org/2001/04/xmlenc#sha256" xmlns="http://schemas.microsoft.com/appx/2010/blockmap">
  - <File LfhSize="42" Size="262144" Name="Registry.dat">
        <Block Size="18709" Hash="ZFMNq9RsFbZyBisZRhu1pJKNwNRhM+ZFEn5ejvCGN8Q="/>
        <Block Size="10578" Hash="wy+YmxV44QRdJxliiOZx+ziemQlwwAIvvYghSJymQ1k="/>
        <Block Size="10530" Hash="OVy9fA0hQ5DuyBJn3p8rjU8zPbVTpknZZFF9ZFtRoY0="/>
        <Block Size="306" Hash="3i8lYGSgr3l3R8K5dQXcC5898N5PSJ6scxwjrpypzDE="/>
    </File>
  - <File LfhSize="93" Size="93088" Name="Root\VFS\AppVPackageDrive\MyVFSFolder\T720-Deco-Regular.ttf.txt">
        <Block Size="31265" Hash="kVNxTvZ0TTkm0PkTqRccu4p5KN+ujyRsyn69POnNfXU="/>
        <Block Size="12314" Hash="Izrttz7xSswoRSKrrvDG3MgZ3oRu46XZNGee9KbbDTk="/>
    </File>
  - <File LfhSize="93" Size="79600" Name="Root\VFS\AppVPackageDrive\MyVFSFolder\T731-Roman-Italic.ttf.txt">
        <Block Size="36596" Hash="VAaYQFPJ/J7qU0rxRCEJEbqxkbHfVCLR+SksPL4wuZI="/>
        <Block Size="7421" Hash="0z4SxGe5IxAopQx4hPbGCc0v7lpjo5wMhN4jot5FhJ0="/>
    </File>
  - <File LfhSize="94" Size="74532" Name="Root\VFS\AppVPackageDrive\MyVFSFolder\T731-Roman-Regular.ttf.txt">
        <Block Size="37353" Hash="GGR+ZdoGWHair+JlkTLEaHG1vTM2VM95jN4jIE7F/UU="/>
        <Block Size="5004" Hash="F72uWz2PvTRbGd5k1sSAWPSaZvqyFdB62TV65wdnNFk="/>
    </File>
  - <File LfhSize="88" Size="46056" Name="Root\VFS\AppVPackageDrive\MyVFSFolder\Tabasco-Bold.ttf.txt">
        <Block Size="24943" Hash="0jmpD7PtYE25POqKuO4wxg+Qi6DN3z84IXmC6UN+K9k="/>
    </File>
```

This information is used by the streaming driver. When a portion of a file is needed, the driver looks at the offset and size of the data needed relative to the uncompressed size and using the information above can determine which block of the file to read in and decompress. Combined with information from the Central Directory, the compressed data can be directly accessed in the App-V file.

The more files present in the archive, the longer the relative file paths are, and the larger the AppxBlockMap file becomes. Given that xml parsing performs poorly, and that this xml file is not placed in the client package cache with the rest of the xml files, I am assuming that the client compiles this data into some kind of binary form and caches it somehow. I have not detected where this might be stored, so a possibility is that remains in memory until flushed out (or reboot).

3 Summary of Where Impacts of Folders and Files Are Felt

Files and folders within a package affect performance in several ways:

- Each file (but not folder) in the package increases the size of the .AppV file central directory map, causes more data to be streamed during the Add-AppVClientPackage step. The file also adds a central directory entry, which may have more overhead than actual file for small files. One individual file probably makes no difference in performance, but a large number of them matters.
- Each file, and each folder, increases the size of the FileysystemMetadata.xml file, which must be streamed and uncompressed during the Add-AppVClientPackage step.
- Each file (but not folder) in the package increases the size of the AppXBlockMap.xml file. Each 64kb (uncompressed) of each file increases this a little bit more. This XML information must be streamed prior to streaming any other data from the App-V file, so this also affects the Add-AppVClientPackage step[6].
- During Publish-AppvClientPackage step, the folder and reparse point for some of the package Folders (but not files) are created in the App-V Cache. The impact of this is not measureable in this test due to the limited number of folders involved.
- During Mount operations, whether command or background streaming, these components, if not already natively in place, are laid down on the native system. This includes creating folders (when not already done) and creating the reparse point, sparse blocks, and streaming down the content. **The creation of these sparse points represents the third biggest slowdown of deployment performance that you can easily avoid (removing unnecessary files) that was found in the testing that produced this research paper series.**
- Additional performance degradation at runtime (launch) was also detected, even when the files/folders are not used. The source of this is unknown at this time.

Unlike when repackaging application installers, the cleanup of unnecessary files, folders, and registry items from a package is not traditionally recommend for App-V sequencing (because we tend to break too much by performing the cleanup). Removal of unnecessary files and folders, however, can improve deployment performance. As will be seen in the companion paper on the Virtual Registry, registry cleanup is still not recommended.

[6] The presence of large file data also affects performance elsewhere, but this is covered in a separate research paper on Big File Performance.

4 Testing Strategy Used

This section provides details about 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 or 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 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.

4.3.1 Test Package

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

- Waiting
- Add-AppVClientPackage
- Waiting
- Publish-AppVClientPackage
- Waiting
- [Optionally Mount-AppVClientPackage[7]]
- Waiting
- First run (launch “cmd.exe[8] /c time /t” inside the virtual environment).
- Waiting
- Second run (launch “cmd.exe[9] /c time /t” inside the virtual environment).

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

4.3.2 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.

[7] 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.

[8] 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.

[9] 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 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

4.3.3 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 achieves 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.

4.4 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.

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 contain no real application. A script was used during sequencing to create the number of folders and small files to be added to the package. Scripts also contain a shortcut to an external cmd prompt for debugging. During sequencing, some background activity by the OS resulted in additional file/registry captures, but the amount of these items is quite small when compared to the number of intentionally added items.

5.1 Warm-up Package

This package is primarily used as the first package in a Test Pass, to warm up the OS and App-V Client components and dependencies[10].

5.2 LotsOfNothing (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).

The package was tested to produce a baseline for "absolute minimum" of what the App-V Client can do. These numbers are useful in determining the amount of overhead that the VC Runtimes place on the system.

[10] When conducting tests that use mounting, I found it necessary to warm up the system without mounting this package. It appears that the first client activity after boot requires additional time to warm up the client, possibly loading drivers. But I also found that mounting this package causes an odd additional 1 second hit to any subsequently Add-AppVClientPackage commands (even after settling time). 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.

5.3 LotsOfFiles_PVAD

This package consists of 50 folders under the PVAD folder, each containing 1000 text files, for a total of 50,000[11] files. Each text file contains two characters ("hi"). These files are not referenced when launching the package.

5.4 LotsOfFolders_PVAD_Empty

This package consists of a package consisting of 50,000 folders under the PVAD folder, each containing no files.

5.5 LotsOfFolders_VFS_Empty

This package consists of 50,000 folders under a VFSd folder placed at the root of the C: drive, each containing no files.

5.6 LotsOfFolders_PVAD_WithOneFile

This package consists of 50,000 folders under the PVAD folder, each containing 1000 text files, for a total of 50,000 folders and 50,000 files. Each text file contains two characters ("hi"). These files are not referenced when launching the package.

5.7 LotsOfFolders_VFS_WithOneFile

This package consists of 50,000 folders under a VFSd folder placed at the root of the C: drive, each containing 1000 text files, for a total of 50,000 folders and 50,000 files. Each text file contains two characters ("hi"). These files are not referenced when launching the package.

5.8 LotsOfFiles_D50_F1000

This package consists of 50 folders with 1,000 file each, all in the PVAD folder. Each text file contains two characters ("hi"). These files are not referenced when launching the package.

[11] The actual number for this package turned out to be slightly smaller due to a scripting bug that generated the package. It was nearly 49,000. This bug did not affect the other packages used.

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!

I created a tool called "AppV File Investigator[12]" to pull out some details of the packages. A summary table of some details about the packages:

	Baseline	Folders (PVAD)	Folders (VFS)	Both (PVAD)	Both (VFS)	Files (PVAD)
Folders	0	50,000	50,000	50,000	50,000	~50
Files	0	0	0	50,000	50,000	~50,000
AppV Size	29,696	145,839	850,231	9,851,815	12,442,985	13,179,735
Directory Block Size	722	740	3,236	4,492,100	6,089,775	6,561,151
FilesystemMetadata Compressed	244	124,377	299,746	267,631	146,988	336,527
FilesystemMetadata Uncompressed	364	1,639,395	5,480,500	5,430,452	4,689,711	3,377,295
AppxBlockMap Compressed	615	1,606	6,619	162,106	188,131	183,304
AppxBlockMap Uncompressed	1,118	3,020	14,425	6,503,220	8,095,392	7,463,147
# Entries	8	8	27	50,027	50,009	48,958
# Empty Folders	0	50,000	50,000	0	0	0
# Opaque Folders	0	0	0	50,000	1	49

Regarding the "Number of entries": The App-V file always contains a minimum of the 6 xml files above the root, plus the virtual registry data file. Our packages have a minimum of 8 entries because they also contain a shortcut link and icon file (created by the sequencer for the shortcut to the cmd prompt)[13].

[12] Available from TMurgent website. This functionality is now directly available in the AppV_Manage tool also.

[13] Some packages also captured additional items which I did not clean up. There was also a scripting mistake made in the 50 folder/50,000 file package creation that reduced the number of files slightly.

From this table we can see that the empty folders are listed only in the FilesystemMetadata xml file, and these do not have actual entries in the directory. Although the compression format does allow for the definition of folders, it seems that App-V chose not to include them as independent entries in the App-V file. Where there is a file, it will have an entry, including its relative path as part of the entry. Those folders without any files underneath them at some level to generate an induced path definition are then called out in the FileystemMetadata file.

ALTHOUGH BASED ON THE ZIP COMPRESSION FORMAT, A COMPLETE FILESYSTEM LISTING MUST HAVE THE EMPTY FOLDERS AND PULLUCIDITY SETTINGS OF THE FILESYSTEMMETADATA.XML FILE ADDED TO THEM.

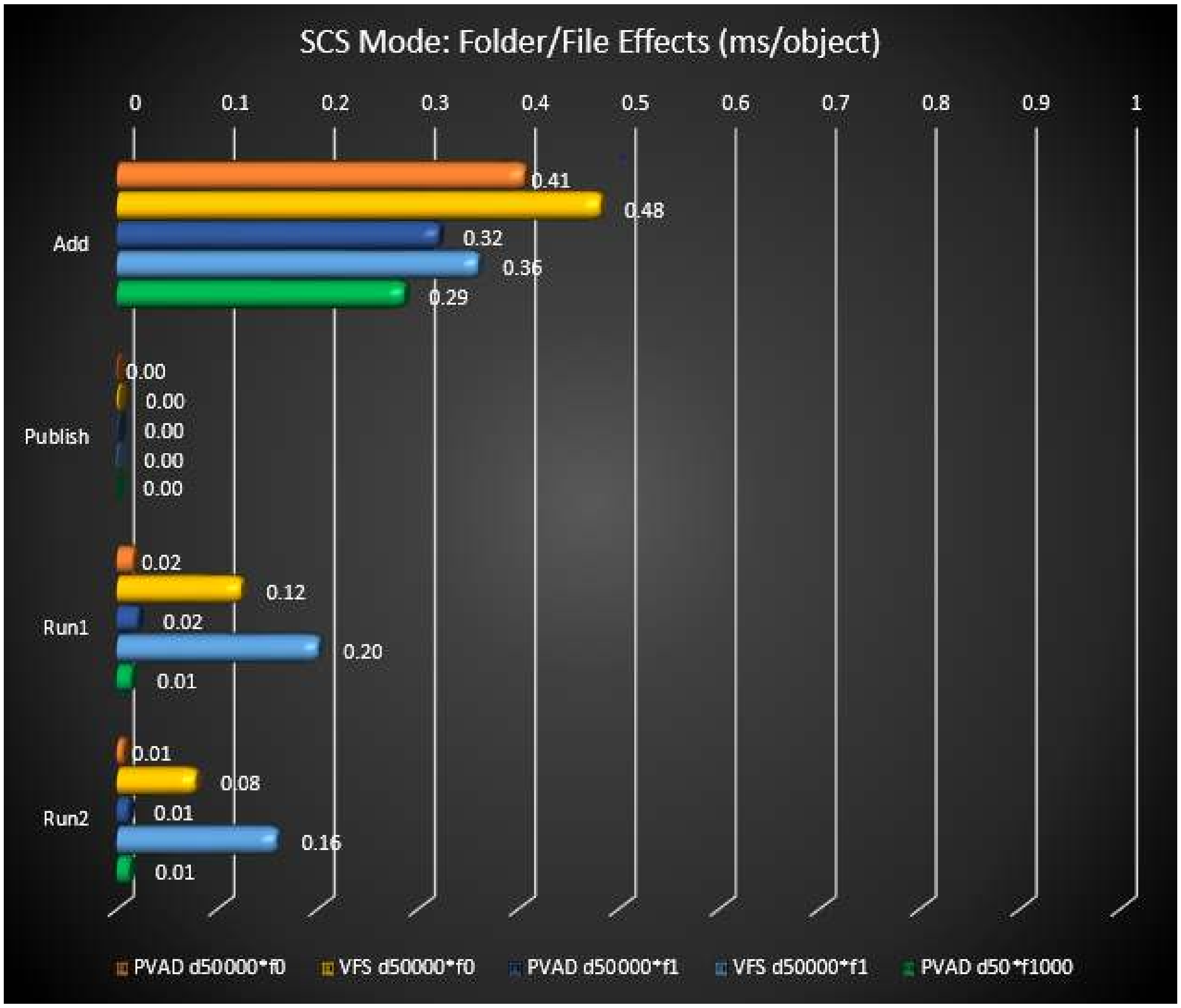
Because the folder is not defined in the directory, there is also no place to store additional attributes. This design choice led the development team to encode the Pellucidity attribute for the folders inside the FilesystemMetadata file as well. These are the “OpaqueDirectories” elements of the FilesystemMetadata file.

This, in part[14], is why you should not attempt to edit an App-V package using a standard zip utility.

[14] The client and management server also check for other “special indicators” to detect outside editing. Ultimately, the stream information in the App-V XML file must reflect compressed record offsets precisely, which would be out of sync if the compressed file is saved using a non App-V aware editor utility.

6.1 SCS Mode Testing without Mounting

Tests were performed with SCS mode enabled to show the impact on the various combinations of unused files and folders (they are not accessed at runtime by the "app"). The difference from the baseline package is given, divided by the total number of file+folder objects.



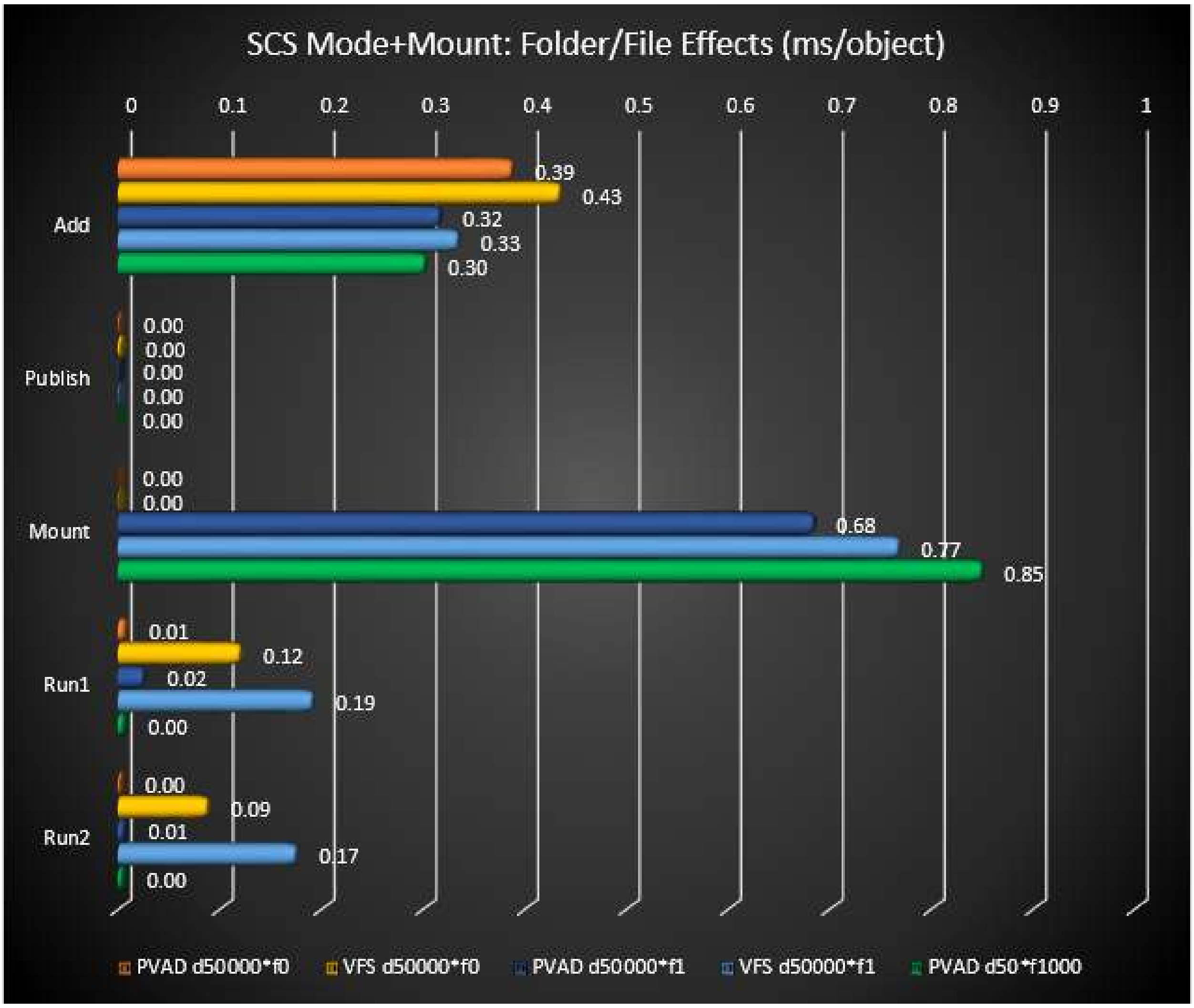
A VFS penalty for folders is seen both at the time of Add package, and at runtime.

The Add Package impact is likely due to the larger FilesystemManifest.xml file and more xml parsing required.

As the files are not being accessed, the cause for increased runtime for the VFS cases is not known. Very surprising is that subsequent launches were longer. This appears to be due to something that changed in HF4 as I do not recall seeing this before.

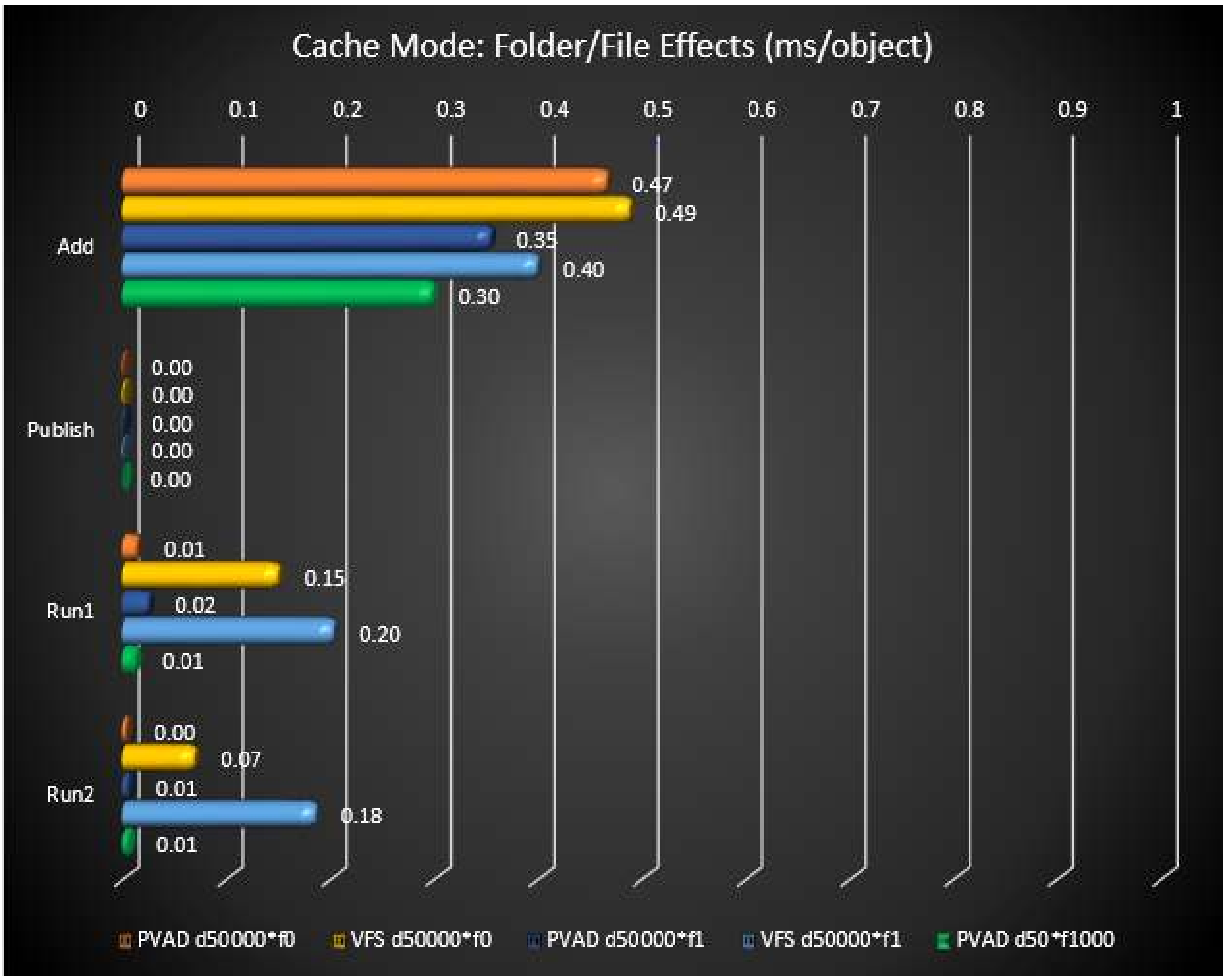
6.2 SCS Mode Testing with Mounting

Tests were also performed with mounting.



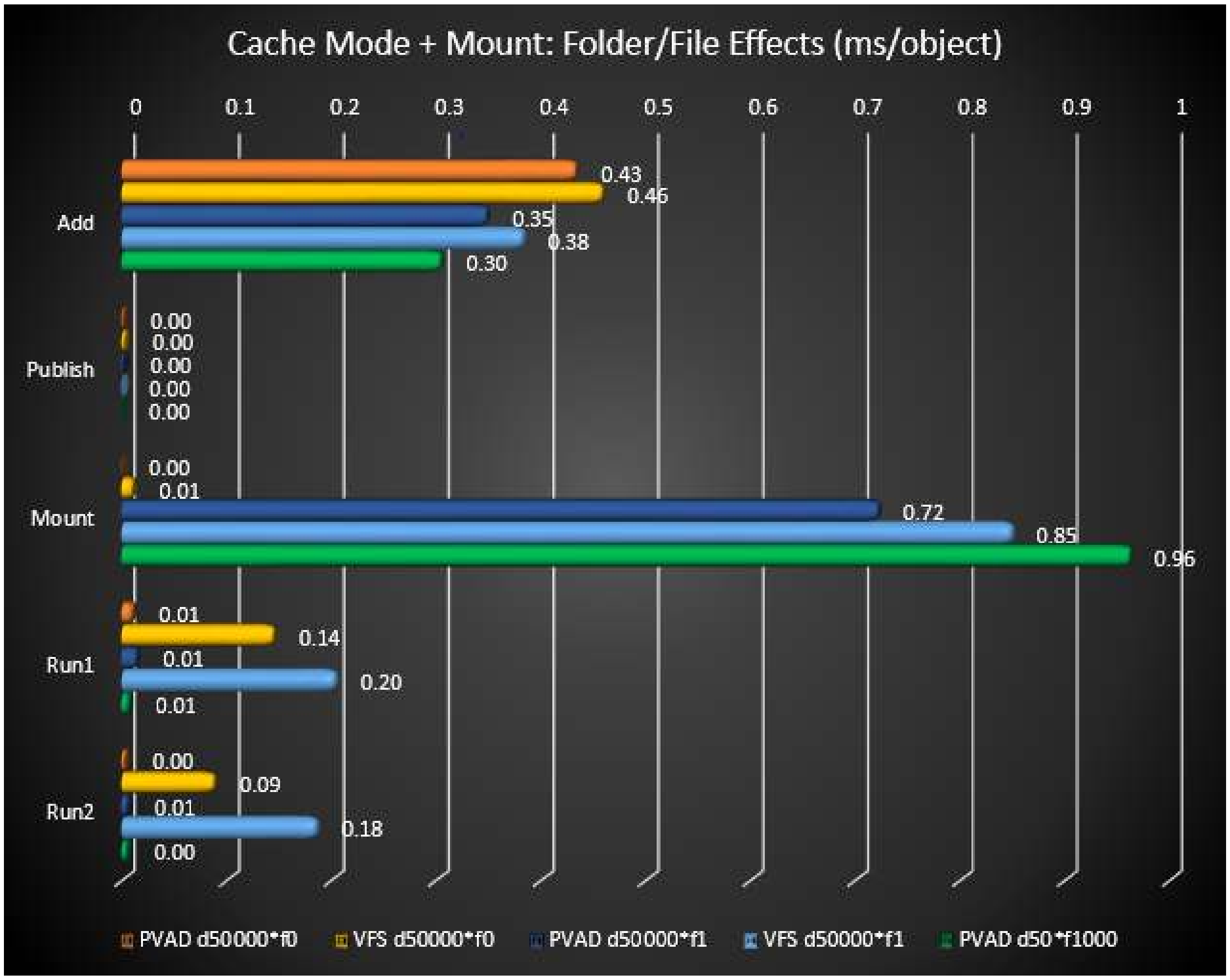
6.3 Cached Mode without Mounting

These are results for running with SCS disabled (and automount disabled).



6.4 Cached Mode with Mounting

Results from cached mode after mounting.



6.5 Impact Analysis

In situations where deployment performance is crucial, such as VDI scenarios, these results show the third most[15] dramatic degradation in deployment performance of any of the tests run in this series.

There are numerous lessons to learn from this data.

6.5.1 Impact of unnecessary files

The addition of a large number of unnecessary files in the package greatly affects Add-AppvClientPackage. The impact hits this step in four parts:

- A larger App-V Central Directory to stream and read.
- A larger AppxBlockMap.xml file to read and parse.
- A larger FilesystemMetadata.xml file to stream and parse.
- More filesystem objects to pre-stage in the App-V cache.

This impact is (with the exception of the impact of increased AppxBlockMap size) unrelated to the size of the file. Calculations[16] from the data allow for an estimate of the overhead for each file to the Add-AppVClientPackage step:

EACH EXTRA FILE ADDS LESS THAN 1/3MS EACH TO THE ADD-APPVCLIENTPACKAGE STEP

These files have a more significant impact to mounting operations, whether performed through the mount cmdlet or via background automatic loading. This was measured in the tests to be about 1ms per small file. Obviously larger files will take longer, but there is a lot of overhead in establishing the first portion of the file.

EACH EXTRA FILE ADDS ABOUT 0.7 TO 0.9MS TO THE MOUNT STEP, PLUS WHATEVER TIME IT TAKES TO STREAM ITS CONTENTS

[15] VC Runtime components being the *most* dramatic.

[16] The difference between 50 folders and 5000 folders with the same number of files indicates that folders containing files have a 0.03ms effect. Therefore the d50f1000 result of 0.29ms is very close to the straight file effect.

There also appears to be a smaller, but measurable, impact during the first run of the package (even though the file is not used) when the file is in the VFS. I did not look into the cause of this impact, but perhaps it was because the files used in the test were all of .txt extensions

Subsequent runs of the package appear to have a negative performance affect also.

EACH EXTRA FILE ADDS ABOUT .05MS TO THE FIRST RUN, AND .10MS TO ADDITIONAL RUNS.

It is relatively easy to identify and remove at least some of the unused files in a package, especially when it comes to some standard things to look for, such as cached copies of installers. Greater gains are likely to be found in larger software packages, however, performing this work would require a good understanding of the specific application and its files.

In some cases, instead of deleting files outright, it might be possible to configure the application to use an external folder for storage of these items (presumably read-only). When the application does not have such a configuration option, it might also be possible to replace the folder inside the package with a directory junction point to an external share to contain the items. Such extreme measures should probably only be taken when there are very large numbers of files and deployment performance in App-V for the package proves unacceptable to users without it.

6.5.2 Impact of Unnecessary Folders

The addition of unnecessary folders also negatively influences deployment performance. Individually the impact is greater than that of an unnecessary file, however, it is unlikely that there are a large number of these unnecessary folders present in a package.

The impact hits this step in three parts:

- A larger AppxBlockMap.xml file to read and parse.
- A larger FilesystemMetadata.xml file to stream and parse.
- More filesystem objects to pre-stage in the App-V cache.

Unlike the file, the folder does not affect the size of the Central Directory. The larger overhead instead likely comes from the client/driver implementing the information about these folders from the FilesystemMetadata file in its memory structures.

EACH EXTRA FOLDER ADDS ABOUT 0.03MS EACH TO THE ADD-APPVCLIENTPACKAGE STEP

At first run, the VFS folders cause an additional impact. The cause of this is unknown at this time.

EACH EXTRA VFS FOLDER ADDS ABOUT .15MS TO THE FIRST RUN, BUT ONLY .07MS TO ADDITIONAL RUNS.

The performance hit on PVAD folders at runtime is only barely measurable.

Empty folders, although probably limited in numbers, are probably safer to remove in most cases. The chances are that these folders were created to hold optional components not present or run-time data. Deleting these folders tends to not break things, because:

- Typical code to look for files inside the folder acts the same whether the file is not present or the folder is not present.
- Typical code to write a file inside the folder will generate the folder when not present.

The latter point depends upon the developer writing reasonable code, but most of the time they will do this correctly.

Identifying the empty folders is also easy to do. Just rename the App-V file, pop open the FilesystemMetadata.xml file and read the list!

6.5.3 The Effect of VFS

It turns out that having things in the VFS, instead of in the PVAD folder, has a much larger impact on deployment (and runtime) performance than I would have expected.

One portion of this deployment effect comes from the things having longer relative paths. This increases the size of the overall App-V file, the size of the Central Directory map, and the size of each of the XML files where things are referenced.

Still, the effect is greater than I can predict from the other behaviors I observed during the testing. Perhaps this is an area for further research someday, especially for the VFS files.

Add-AppvClientPackage:	.07ms/folder,	.004ms/file
First Run:	.15ms/folder,	.37ms/file
Subsequent Run:	.06ms/folder,	.35ms/file

At this point, this information is probably not actionable. There will be a few packages that install an overly large component into the VFS (for example Business Objects). And while it might be tempting to just designate that folder as the PVAD and let the main, smaller, application be VFSd to improve deployment performance, you stand a good chance of having other issues, for example if you use connection groups. Pulling that VFS information into a separate package might be considered, but the overall performance hit of deploying another package far outweighs the small impacts reported above.

7 About This Research Paper Series

This research paper is part of a series of papers, released by TMurgent Technologies, which 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