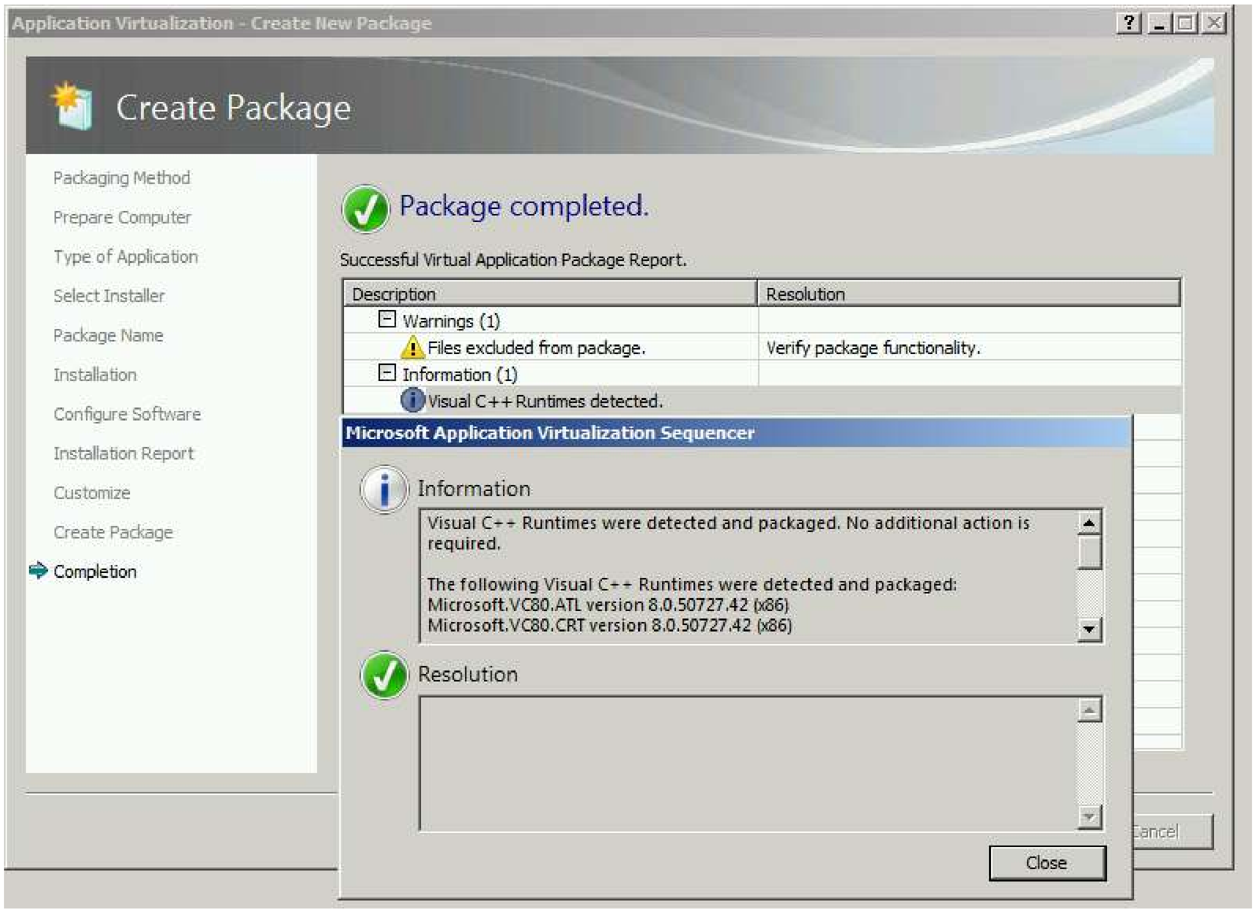
Effects of “VC RunTimes” in App-V 5 SP2 with HotFix 4 Deployment Performance

TMurgent App-V 5 Performance Research Series



Package completed.

Successful Virtual Application Package Report.

Description	Resolution
Warnings (1)	
Files excluded from package.	Verify package functionality.
Information (1)	
Visual C++ Runtimes detected.	

Microsoft Application Virtualization Sequencer

Information

Visual C++ Runtimes were detected and packaged. No additional action is required.

The following Visual C++ Runtimes were detected and packaged:
Microsoft.VC80.ATL version 8.0.50727.42 (x86)
Microsoft.VC80.CRT version 8.0.50727.42 (x86)

Resolution

June, 2014

Contents

1 Introduction

The purpose of this research paper is to document the effects that certain detected side-by-side components (certain VC Runtime and MSXML components) have in Microsoft App-V Virtual Application Packages.

The effort is squarely aimed at answering questions on how the detection/deployment of these components in a 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 VC Runtimes

Microsoft made a change to the OS platform at some point, maybe 3 or 4 years ago, that caused certain[1] side-by-side components to break virtual applications if they were captured inside the sequence. The source of this change is unknown, but we don't think it was an App-V change, and packages that once worked stopped working in App-V 4.6 and above.

We initially noticed this on some VC Runtimes, but soon discovered that Microsoft MSXML also suffered this fate. I also include Direct-X in the list of things suddenly broken, but there might be a different cause on that one.

In 4.6, we developed a practice of keeping these components out of the package, and making sure that we installed all of them on the base OS image used at the clients. I maintain a couple of spreadsheets on my website that contain information relating each of the VC Runtime and MSXML version files with links to the download page from Microsoft to get the installer.

Getting all of them installed on the base OS image was not that difficult when a company has a good imaging process, or at least a good native deployment capability like SCCM.

Getting all of them installed the way we needed to on the sequencer was another story. We found that some of the installers, especially VC Runtime service packs and security updates, actively remove certain older versions known to contain a vulnerability. If this happened on the sequencer snapshot and then we install an application that installed the "old" version, then the old version got captured inside the package and we again had a broken package. So while we would want to install on the client oldest to newest, on the sequencer snapshot it needed to be newest to oldest.

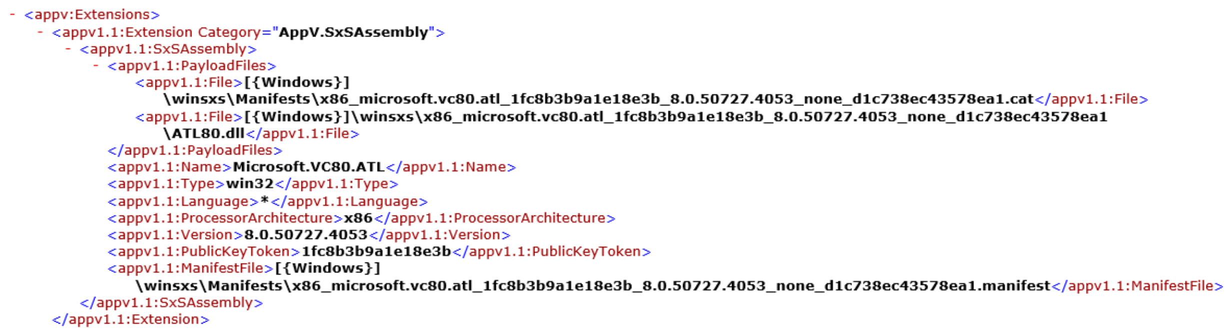
This continued to be a problem in App-V 5.0, and Microsoft developed a solution in 5.0 SP2.

[1] See http://www.tmurgent.com/appv/index.php/resources/tools/167-visual-studio-runtime-versions-chart and http://www.tmurgent.com/appv/index.php/resources/tools/157-msxml-versions-chart for lists of versions that cause issues in App-V.

2.1 Detecting VC Runtimes/MSXML In A Package

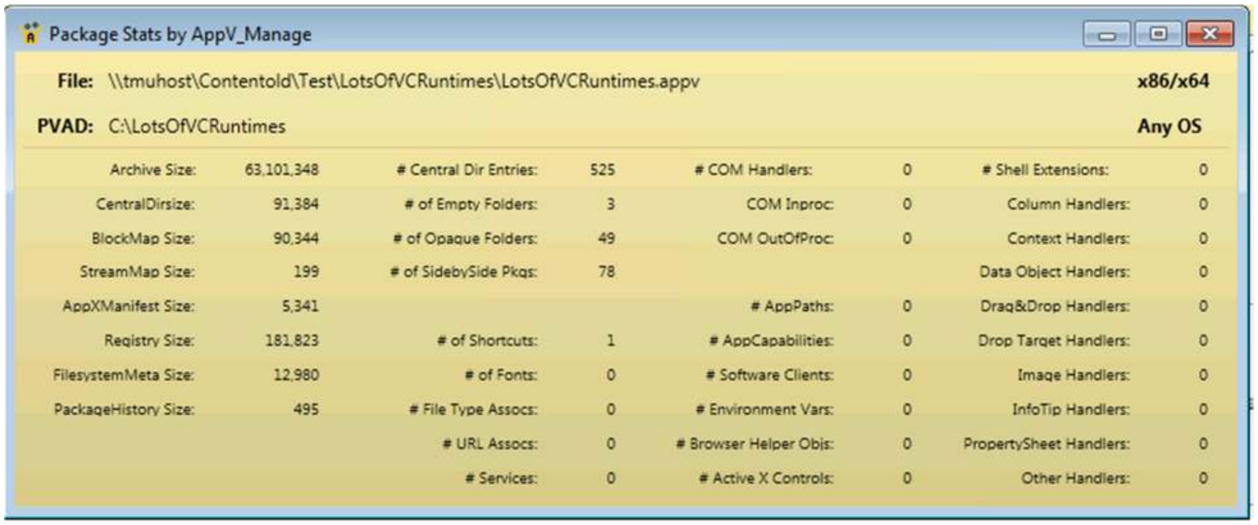
Starting in 5.0 SP2, Microsoft made changes to detect and support these components when virtualized.

While there was detection logic added to the sequencer in 5.0, it appears that this logic simply detects and calls out the side by side components in the report and lists them in the internal AppX_Manifest file.



```
- <appv:Extensions>
  - <appv1.1:Extension Category="AppV.SxSAssembly">
    - <appv1.1:SxSAssembly>
      - <appv1.1:PayloadFiles>
          <appv1.1:File>[{Windows}]
            \winsxs\Manifests\x86_microsoft.vc80.atl_1fc8b3b9a1e18e3b_8.0.50727.4053_none_d1c738ec43578ea1.cat</appv1.1:File>
          <appv1.1:File>[{Windows}]\winsxs\x86_microsoft.vc80.atl_1fc8b3b9a1e18e3b_8.0.50727.4053_none_d1c738ec43578ea1
            \ATL80.dll</appv1.1:File>
        </appv1.1:PayloadFiles>
        <appv1.1:Name>Microsoft.VC80.ATL</appv1.1:Name>
        <appv1.1:Type>win32</appv1.1:Type>
        <appv1.1:Language>*</appv1.1:Language>
        <appv1.1:ProcessorArchitecture>x86</appv1.1:ProcessorArchitecture>
        <appv1.1:Version>8.0.50727.4053</appv1.1:Version>
        <appv1.1:PublicKeyToken>1fc8b3b9a1e18e3b</appv1.1:PublicKeyToken>
        <appv1.1:ManifestFile>[{Windows}]
          \winsxs\Manifests\x86_microsoft.vc80.atl_1fc8b3b9a1e18e3b_8.0.50727.4053_none_d1c738ec43578ea1.manifest</appv1.1:ManifestFile>
      </appv1.1:SxSAssembly>
    </appv1.1:Extension>
```

As the internal manifest is not readily available, the AppV_Manage tool (a free tool from TMurgent) provides this information as well. On the publishing tab, click on the package and then on the Analyze button to see if any are present in the manifest. Hover the mouse over the counter for a list of the packages detected.[UPDATE GRAPHIC WITH TOOLTIP]



Package Stats by AppV_Manage

File: \\tmuhost\Contentold\Test\LotsOfVCRuntimes\LotsOfVCRuntimes.appv **x86/x64**

PVAD: C:\LotsOfVCRuntimes **Any OS**

Archive Size:	63,101,348	# Central Dir Entries:	525	# COM Handlers:	0	# Shell Extensions:	0
CentralDirsize:	91,384	# of Empty Folders:	3	COM Inproc:	0	Column Handlers:	0
BlockMap Size:	90,344	# of Opaque Folders:	49	COM OutOfProc:	0	Context Handlers:	0
StreamMap Size:	199	# of SidebySide Pkgs:	78			Data Object Handlers:	0
AppXManifest Size:	5,341			# AppPaths:	0	Drag&Drop Handlers:	0
Registry Size:	181,823	# of Shortcuts:	1	# AppCapabilities:	0	Drop Target Handlers:	0
FilesystemMeta Size:	12,980	# of Fonts:	0	# Software Clients:	0	Image Handlers:	0
PackageHistory Size:	495	# 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

The important changes to support these components are located at the App-V 5.0 SP2 and above Clients. In fact, the solution works with packages sequenced prior to SP2 and happen to have the components present in the file system.

Microsoft's client implementation detects the component during deployment operations, extracts them from the package and adds them natively to the system.

At first, I was glad not to have to deal with these components any more. We could just sequence and deploy and never worry about them again. But I worried.

My initial worry was that we could capture a VC Runtime in a package that later turns out to contain a vulnerability. In a situation without App-V, installing the new version of the VC Runtime fixes this by actively removing the vulnerable version. When the app requests the old version, the replacement is in the same family and is supplied by the OS instead and all is well.

With 5.0 SP2, if I publish the package, and then later install the updated VC Runtime, the same thing happens. This is good.

But, if another user receives the VC Runtime patch first to their base image and later gets the App-V package, the vulnerable version of the VC Runtime is laid down and used by the package. Not a good thing! So when the feature was released I cautioned customers about this. Many couldn't care less, but for some security conscious customers this behavior was unacceptable and they decided to retain the practice of excluding them from the package.

Now with the performance results described in this paper, we have a second reason to keep these components out of our packages!

3 Summary of Where Impacts of VC Runtimes Are Felt

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

VC Runtimes/MSXML affect deployment performance in several ways:

- An increase in size of the AppX_Manifest file, where the detected components are recorded, affect the Add-AppVClientPackage to a small extent. This file is copied locally and parsed as part of this process.
- As additional files are in the package, these slow up add and publish operations. Add-AppVClientPackage is further slowed down as the directory of the .AppV file is larger. Publish-AppVClientpackage is further slowed down as the sparse blocks for these components must be created (subject of a different research paper in this series) and streamed down.
- During Publish-AppVClientPackage, these components, if not already natively in place, are laid down on the native system. This is done by the client component (avoiding elevation issues) and uses the "trustedinstaller" process on the individual files (rather than the component installer). **This represents the single biggest slowdown of deployment performance that you can easily avoid found in the testing that produced this research paper series.** Simply pre-installing the VC Runtimes cuts the impact by over half, but pre-installing and removing from the package is recommended for optimum deployment performance.
- Additional performance degradation at runtime was also detected. The source of this is unknown at this time.

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[2]]
- Waiting
- First run (launch "cmd.exe[3] /c time /t" inside the virtual environment).
- Waiting
- Second run (launch "cmd.exe[4] /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.
- 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.

[2] 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.

[3] 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.

[4] 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 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 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.

4.4 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.

5.1.1 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[5].

5.1.2 FullOfNothing (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.

5.1.3 LotsOfVCRuntimes

This package consists of all of the VC Runtimes that I had available, from 2005 through 2012, installed from oldest to newest. While this consisted of 13 installers, the end result contained 78 VC Runtime components in the final report.

[5] 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.

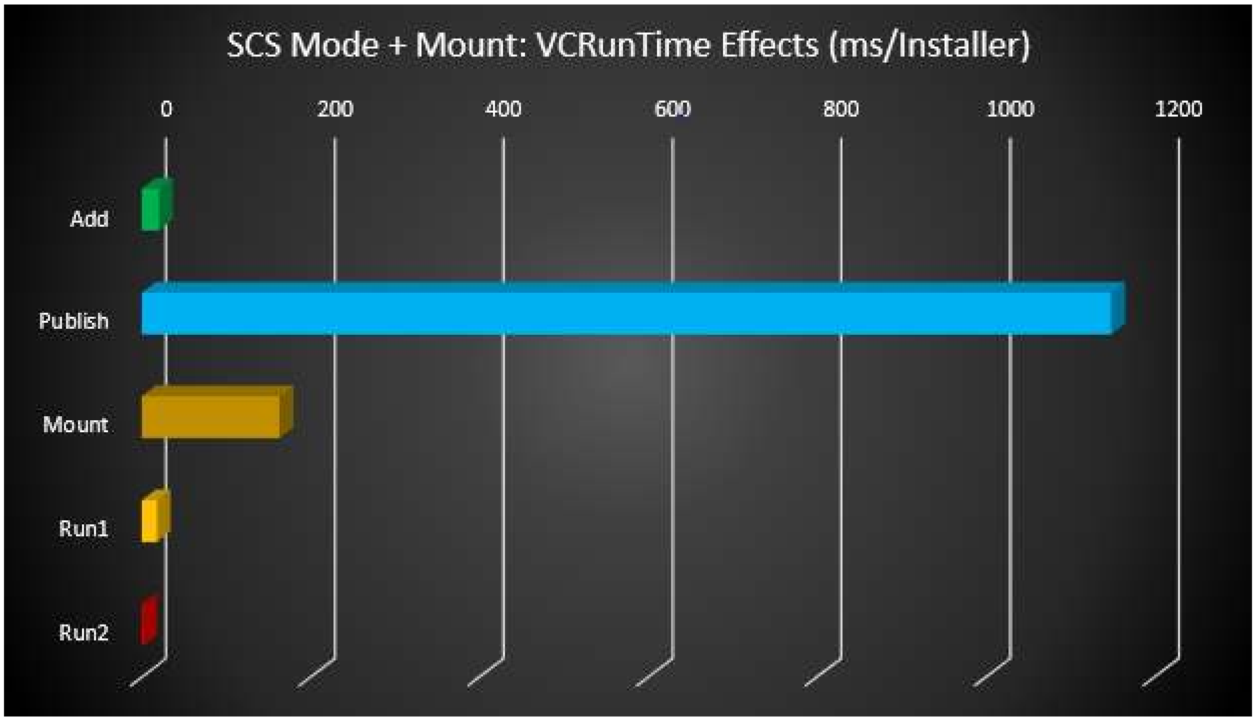
6 Detail Test Results

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!

Tests were performed with and without Mounting, and with SCS Mode enabled or disabled. The package was tested two different ways:

- On a "clean" client that contained only those VC-Runtimes required by the App-V Client. Because the impact is primarily on publishing, the results of all scenarios are similar.



- On a client with all of the VC Runtimes pre-installed in the same order as used in the package. These tests reduced the publishing time to less than half of the time.

In situations where deployment performance is crucial, such as VDI scenarios, these results show the most dramatic degradation in deployment performance of any of the tests run in this series. And it is the easiest change in your approach to achieve improved performance, roughly 200ms per captured file (1.14 seconds per VC installer package) by removing the component you're your package and pre-installing on the client image.

From the numbers we reach the following conclusions:

Each captured VC Runtime Installer adds about 20ms Add step (3.6ms/VC File)

Each captured VC Runtime Installer adds about 1.15sec to the Publish step, unless pre-installed (204ms /VC File)

Each captured VC Runtime Installer adds about 18ms to the First Run (unless pre-installed)

Once installed, the publishing time due to VC Runtimes is reduced by more than 50%.

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