PDS: A Virtual Execution Environment for Software Deployment

Bowen Alpern, Joshua Auerbach, Vasanth Bala, Thomas Frauenhofer,
Todd Mummert, Michael Pigott
{alpernb, josh, vbala, tvf, mummert}@us.ibm.com
IBM Thomas J. Watson Research Center
Hawthorne, NY

ABSTRACT

The Progressive Deployment System (PDS) is a virtual execution environment and infrastructure designed specifically for deploying software, or "assets", on demand while enabling management from a central location. PDS intercepts a select subset of system calls on the target machine to provide a partial virtualization at the operating system level. This enables an asset's install-time environment to be reproduced virtually while otherwise not isolating the asset from peer applications on the target machine. Asset components, or "shards", are fetched as they are needed (or they may be pre-fetched), enabling the asset to be progressively deployed by overlapping deployment with execution. Cryptographic digests are used to eliminate redundant shards within and among assets, which enables more efficient deployment. A framework is provided for intercepting interfaces above the operating system (e.g., Java class loading), enabling optimizations requiring semantic awareness not present at the OS level. The paper presents the design of PDS, motivates its "porous isolation model" with respect to the challenges of software deployment, and presents measurements of PDS's execution characteristics.

Categories and Subject Descriptors

K.6.2 [Management of Computing and Information Systems]: Installation Management; K.6.3 [Management of Computing and Information Systems]: Software Management.

General Terms

Management.

Keywords

Virtualization, deployment, management, installation, streaming.

1. INTRODUCTION

Virtual machines, particularly those that attempt to capture an entire machine's state, are increasingly being used as vehicles for deploying software, providing predictability and centralized control [14][21][22][30]. The virtual environment provides isolation from the uncontrolled variability of target machines, particularly from potentially conflicting versions of prerequisite software. Skilled personnel assemble a self-contained software universe (potentially including the operating system) with all of the dependencies of an application, or suite of applications, correctly resolved. They then have confidence that this software will exhibit the same behavior on every machine, since a virtual machine monitor (VMM) will be interposed between it and the real machine.

VEE'05, June 11–12, 2005, Chicago, Illinois, USA.

Because software deployment is a relatively new motivation for using virtual machine technology, today's VM-based software deployment efforts employ VMs that were originally designed for other purposes, such as crash protection, low-level debugging, process migration, system archival, or OS development, and are being re-purposed for software deployment. This paper explores the characteristics of a virtual execution environment that was designed from the start as a software deployment vehicle.

1.1 Why Deployment is Complicated

Consider a scenario in which several different applications produced by separate organizations need to be integrated on the same machine. An example of such a scenario could be a suite such as MySQL[10]/JBOSS[12]/Tomcat[5]/Apache[1], a Java development tool such as Eclipse, and a J2EE application that needs to be developed using Eclipse and tested on the MySQL/JBOSS/Tomcat/Apache suite.

1.1.1 Conflicting Pre-requisites Stimulate Code-Bloat

A complex collection of applications will inevitably have conflicting pre-requisites. For instance, each application may require its own version of the Java Virtual Machine, or depend on specific patch-levels of certain dependent components.

Virtual machine monitors can help tame such conflicts by allowing each application's dependencies to be embedded in its private VM image. To understand why a more specialized kind of virtualization is needed, first consider that vendors are *already* attempting to deal with dependency conflicts in more or less the same way.

Increasingly, vendors are trying to reduce dependency conflicts by embedding the application's dependencies into the application installed image, usually without the benefit of VM technology. For example, Eclipse versions 2.x and above come bundled with Tomcat, which is used for rendering the Eclipse help pages; similarly JBoss distributions also include an embedded Tomcat version. Many commercial Java middleware products sold today embed one or more Java Virtual Machines in their images. This trend is even reflected within a single software product. For

example, the module org.apache.xerces is often duplicated in several different components in an effort to isolate these components more fully from one another. What a VMM adds is a hard guarantee that the isolation between conflicting software stacks is "provably complete," lacking in subtle holes.

But, whether assisted by a VMM or not, incorporation of dependencies without any compensating measures results in increasing software bloat. From a disk space perspective, one could argue that tolerating such bloat is no big deal. But an isolation strategy accomplished through physical code duplication creates other problems. It slows down the deployment process, and increases the number of components that need to be configured at deployment time, or touched during subsequent updates. It also increases the customer's perception of an application's complexity, which in turn increases customers' reluctance to update frequently. This results in a proliferation of software versions in the field and increasing support and services costs over time.

Also, data center environments are increasingly moving toward a "scale-out" model where large farms consisting of several thousand commodity servers are becoming commonplace. In such scenarios, hardware failures can occur frequently, often several times a day. The cost of commodity hardware is now so low that operators deal with hardware failures by simply replacing the defective machine on a rack, and re-provisioning the new machine with the application suite. Large commercial software stacks can take hours to provision, thus increasing the cost of such failures.

Using any VMM to help with provisioning will undoubtedly speed this up by replacing the normal installation process with an easily-moved image. But, unless specific engineering steps are taken to deal with the underlying code bloat, just the process of moving the bits will cause significant slowdown. Again, reversing the trend toward increasing bloat due to duplication-based isolation techniques would be valuable in such situations. And, a properly engineered solution will also take into account that a software application can usually begin executing when only a fraction of its bits are present.

1.1.2 Multi-sourced Software Still Needs to Cooperate

A software deployment system must assume that the software it deploys in one offering is not the only software offering deployed on the target machine. Each machine owner assembles a palette of offerings that suits his or her needs. These offerings must be able to interoperate both via system-mediated communication channels (e.g., semaphores, pipes, shared memory) and via files in a common file system.

Consider the implications for a VMM-assisted deployment. If all offerings were run in the same VM instance, the isolation advantages of using a VM will be lost since the offerings might then conflict. But, if each offering is run in a different VM instance using the usual hardware virtualization paradigm, the interoperation between offerings takes on characteristics of inter-machine communication rather than intra-machine communication. What seems like one machine to the user is now laced with "remote" file mounts and "distributed" protocols. Somehow, the degree of isolation must be relaxed to permit a more local style of interoperation. The relaxation must be done

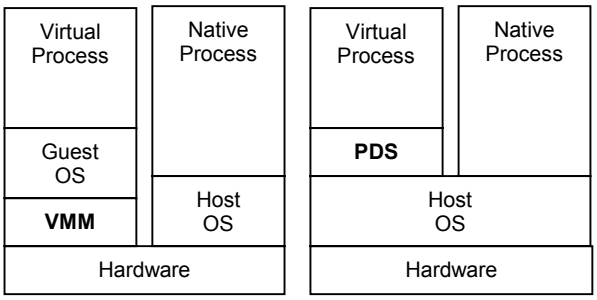

Figure 1 *VMM execution stack compared to PDS*

while still managing conflicts and reducing variability in the areas that matter to correct execution.

Making this change involves a tradeoff. A more "porous" isolation between VMs enhances the user experience when integrating software on a single machine. However, other characteristics that one might expect from a general-purpose VMM (such as crash protection or the ability to freeze and migrate processes) might be sacrificed.

1.2 The PDS Approach

The Progressive Deployment System (PDS) provides a virtual environment for executing *assets* --- self-contained software universes in which all dependencies, *except dependencies on the underlying operating system and hardware*, are resolved.

Assets are designed to be deployed *progressively*, meaning that the transfer of the asset's bits to the target machine is overlapped with its execution. This enables, for instance, replacement racks on a server farm to be rapidly provisioned, without waiting for an entire system image to be moved to the machine prior to starting its execution.

Assets are isolated from each other in the sense that each one sees its own unique resources --- virtual files, directories, and system metadata --- and not resources that are unique to some other asset. While assets cannot see any of the host machine's resources that were overlaid by their own virtual ones, they can see other local resources and can communicate with other programs running on the host machine (including other assets running under PDS) through the local file system and local IPC. The PDS virtualizer puts its assets on the same plane as ordinary programs by running above the OS rather than below it, (see Figure 1). As consequence, however, PDS assets cannot include device drivers and other kernel extensions.

Assets are logically immutable entities, thus ensuring that every asset, once tested, will not later fail due to an incompatible update. Any change to an asset, no matter how small, results in a new asset (as shown in the "virtual view" in Figure 2).

Without an effective mechanism for reducing redundancy between (as well as within) assets, the proliferation of virtual views would entail a prohibitive amount of space to store, and bandwidth to transport, many closely related assets (the "code bloat" problem mentioned previously). To address this difficulty, assets are partitioned into *shards,* variable-sized semantically determined "pages" that are the unit of transfer between a software repository and the host machine. Shards may correspond to files,

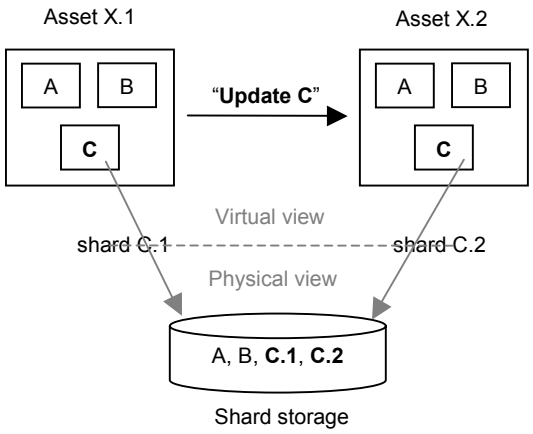

Figure 2 *Virtual vs physical view of software deployment. The fact that the two versions of Asset X differ in component C is only reflected in the underlying physical shard storage.*

semantically cogent portions of files, system metadata such as registry entries, or metadata used by PDS itself. As shown in the "physical view" in Figure 2, shards are freely shared across assets. Bitwise identical shards are given the same physical name (in shard storage) and are only stored once. A reference to C from asset X.1 is mapped to a different shard (shown as shard C.1 in Figure 2) than a reference to C from asset X.2 (shown as shard C.2) while references to A in either asset are mapped to the same shard.

Shards help maintain an appropriately scaled working set as the repertoire of assets in use on a machine evolves over time. Most significantly, since they are semantically determined, they allow redundant parts of highly similar assets to be detected and shared *transparently* (while maintaining the illusion that each asset has its own copy). Thus, the duplication implied by the virtual view of an asset's evolution is not reflected in its physical storage manifestation.

The separation between virtual and physical views of asset composition also enables a software vendor to hide the internal structure of the asset (e.g. the fact that asset X contains components A, B and C) from the end-user. The end-user need only see whole assets (asset X.1, asset X.2, etc.), and never need deal with lower level component patches, upgrades, versions, and configurations. Thus, end-users simply execute the whole asset version they are interested in, and the additional shards required for its execution will be transported automatically.

PDS currently exists as a prototype supporting the Windows OS, but was designed to extend to other operating systems. The prototype has been successfully used to deploy commercial developer tools such as Eclipse [6] and WebSphere Studio [31], productivity environments such as Open Office [18] and Lotus Workplace Client [16], and server stacks like the Apache web server [1] and the Tomcat servlet engine [29].

The balance of the paper is organized as follows. Section 2 details the design of PDS with emphasis on its virtual execution environment. Section 3 presents some measurements of PDS's execution characteristics using assets derived from different versions of Eclipse, Tomcat, and Java runtimes. Section 4 reviews related work. Section 5 discusses future work and Section 6 concludes the paper.

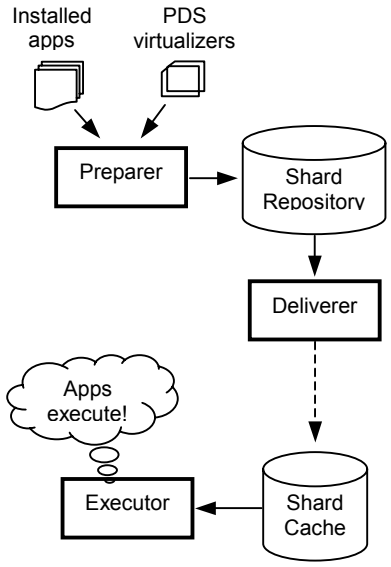

Figure 3 *PDS system overview*

2. PDS SYSTEM DESIGN

PDS is organized into three major components (see Figure 3).

1. The *preparer* produces assets from software that has been installed in the conventional fashion on a "clean" machine (real or virtual) dedicated to the purpose.
2. The *deliverer* makes assets present on a host machine by ensuring the appropriate shards are on hand when needed.
3. The *executor,* PDS's virtual execution environment, manages the execution of assets on the host machine.

We discuss the preparer and deliverer first to provide necessary context.

2.1 The Preparer

The preparer accepts as input an *asset image* and some instructions and it produces a *shard repository* and a *launch document* as output.

2.1.1 Asset Images

To obtain an asset image, one starts with a machine in a known state (ideally, immediately after the OS was installed). Next, one runs conventional installation scripts to install all the software components that make up the asset. Finally, one identifies everything that was added to the system by that series of installations. Typical additions consist mainly of directories and files. However, they may include updates to various system databases, such as the registry in Windows, in which metadata is stored. The additions form the asset image.

2.1.2 Preparation Instructions

The instructions given to the preparer consist of an inventory of what is in the asset image plus a *startup directive.* Typical asset

image inventories consist of just a few directory/file trees (hereafter, "file trees"), but other kinds of system metadata may be listed. The startup directive is a command that executes on the target machine, but inside the virtual environment, in order to start the asset. Most PDS assets have trivial startup directives, but others use the startup directive to set environment variables or perform environment preparation not covered by the asset image inventory.

2.1.3 Shard Repositories

A *shard repository* is a file tree within which each shard is a file. To PDS, a shard has no structure: it is just bits. The cryptographic hash [24] of a shard, called its *shard digest*, is assumed to uniquely identify the shard[1]. In a shard repository, the path names of shards can be algorithmically derived from their shard digests for efficient retrieval. Bitwise-identical shards are only stored once.

The shard repository produced by the preparer contains all the shards of one or more assets under preparation. *Primary* shards are pieces of the original asset image. *Metashards* contain control information generated by the preparer and interpreted by the executor.

The property that bitwise-identical shards are only stored once, has the advantage of automatically avoiding the redundancies implied by every asset containing all of its dependencies. The contents of two virtual files that share the same bit pattern will be represented by the same shard. These files can however have different names, creation dates, permission attributes, etc. PDS reconciles this by storing file metadata in the metashards, and have the primary shards contain only the file contents.

The redundancy avoidance enabled by the shard design also allows separately prepared repositories to be merged to form larger ones containing the shards of many assets but still storing each shard at most once. The deliverer (and sometimes the executor) reads shards from shard repositories but does not mutate them in place (as will be seen, the executor implements copy-on-write semantics for objects in the asset image).

Every primary shard of an asset is referred to in at least one of the asset's metashards via its shard digest, and the metashards themselves form a tree linked by shard digests. The digest of the metashard at the root of this tree uniquely identifies the asset and is called the *asset id* (Figure 4 shows a simplified view of this tree structure as an illustrative example). Any change to an existing asset will produce a new asset with a different id. Thus, all assets are immutable once prepared, but some assets may represent evolutions of others.

2.1.4 Launch Documents

A launch document is a small document (not a shard) containing the asset id of an asset together with additional information that allows the executor to interact with the deliverer to obtain the shards of the asset. For example, this information may specify the location of a shard repository containing the asset's shards.

2.2 The Deliverer

PDS is designed so that the deliverer is readily replaceable and may have a role in the overall system ranging from very large to almost trivial. The interaction between the deliverer and the executor is typically file-based (although small shards can also be read directly into memory). When the executor identifies the need for a particular shard, it passes the shard's identity to the deliverer, which blocks the calling thread until it is able to manifest the shard as a file, at which point the path name of that file is returned to the executor. The executor then uses standard OS interfaces, including memory-mapping, to utilize the shard. The executor does not modify the shard.

Because the shard repositories are just file trees, a deliverer can use file system capabilities already present in the OS to map shard repositories into the local file space. It can employ physical media such as DVDs, it can copy shard repositories to local disk, or it can mount them as remote file systems. The problem of actually moving the bits is left to the file system technology employed. The deliverer simply returns paths in the appropriate file system for each shard requested.

Alternatively, the deliverer can employ a specialized client-server algorithm to transfer shards from a remote shard-repository to a local *shard cache* that contains only those shards needed on the local machine. In this case, the deliverer can implement sophisticated working set maintenance algorithms and pre-fetching of shards based on learned execution patterns. It may also reorganize its shard repositories into alternate representations that do not use a file per shard, for efficiency.

A separable delivery subsystem enables alternative implementations to be plugged in that may be suitable for specific situations. A remote file system that provides good caching and predictive fetching might obviate the need for a specialized deliverer. The current PDS system uses two deliverers, one file based, the other using the HTTP protocol and a standard servlet engine. The latter allows us to experiment with the pre-fetching strategies and operate in wide area networks without requiring the installation of specialized file system software. Pre-fetching results are not presented in this paper but are discussed under future work.

2.3 The Executor

The executor consists of a small bootstrap mechanism to launch the asset on the client system, and the code to provide the virtual execution environment. As will be seen, this code is divided into several *virtualizers*, each with its own task.

PDS provides a virtualization that is both *selective*, to permit assets to interoperate with other local applications via system APIs, and *hierarchical* (in a sense to be explained), to obtain a close mapping between meaningful semantic units and shards.

As previously described, PDS works by interposing a virtualizer between the application and the OS. Exactly how the interposition is accomplished will vary from OS to OS, and there may be alternative strategies for some OS's. The current PDS

[1] While a cryptographic hash is not a unique identifier in the mathematical sense, one can be chosen to make the probability of collision less than the non-zero probability that a disk read will deliver data from the wrong sector. The mathematical justification for cryptographic hashing is beyond the scope of this paper. However, it is a widely used mechanism commercially, and a recommended U.S. government standard.

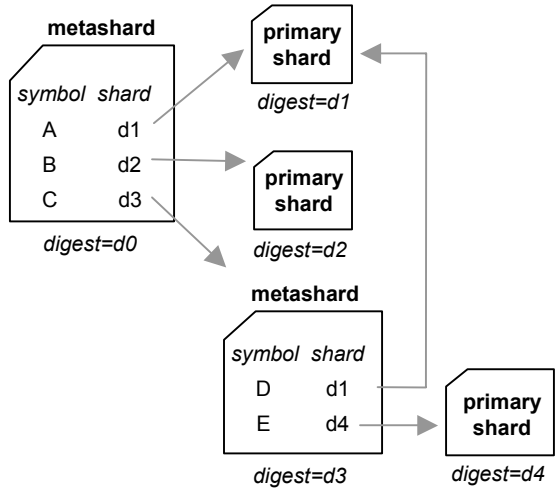

Figure 4 *Metashards and primary shards. The asset id in the example is d0, the digest of the metashard at the root of the tree.*

prototype runs on Windows XP by intercepting a subset of the APIs provided via the **kernel32** and **advapi32** libraries.

2.3.1 Selective Virtualization

The APIs that are intercepted in PDS's selective virtualization are just those needed to map the original asset image onto the target machine as a *virtual asset image* (VAI). That is, they include the APIs that deal with files, directories, system metadata, and anything else that is found to be stored persistently at installation time.

Although the bulk of these APIs are file-related, some asset images include information stored in specialized system databases not accessed via the file APIs (e.g., the Windows registry). Asset images may also include scattered files in system-managed directories, a pattern that cannot be duplicated via the hierarchical mounting capabilities of most operating systems. Finally, dynamic loading and dynamic binding between modules, although rooted in file I/O, has semantic details (search paths, versioning, etc.) that require additional intervention to ensure that the asset operates within its VAI and isn't contaminated by artifacts in the real system. These subtleties make it difficult to accomplish the kind of deployment PDS enables through alternative approaches such as remotely mounting the asset image file trees directly on the host machine.

PDS only intercepts a small subset of the full Windows API, limiting its interception to certain file-related APIs, registry APIs and those related to dynamic loading and process creation. All of the graphics, interprocess communication, network I/O, thread synchronization, and message formatting APIs are left alone, causing a PDS asset to be, in most respects, a peer of other programs running on the OS.

Even within the file APIs, we distinguish between *path directed requests,* in which files are designated by hierarchical names, and *handle directed requests,* in which files are designated by previously opened handles. As is the case with many distributed file systems, we intercept the former but (usually) not the latter, performing the necessary actions (including copying if necessary) at open time to avoid having to interfere with reads, writes, seeks, locking and synchronization. This is done not only for efficiency but also to permit the memory-mapping APIs of the OS to operate without the need for a fine-grained intervention by the virtualizer. Section 2.3.3 provides more details on how the VAI is controlled by the virtualizer.

For those APIs that are intercepted at all, the virtualizer makes a rapid decision based on the path name, registry key, etc. as to whether the request falls within the VAI or not. If it does, the request is handled as discussed in section 2.3.3. But, if not, the request is passed through unchanged to the operating system. Thus, PDS assets can communicate via the local file system with each other, with non-PDS programs, and with OS utilities.

2.3.2 Hierarchical Virtualization

Intercepting only at the OS level limits a virtualizer's ability to optimize the storage and transfer of shards representing files. Files can be arbitrarily large, with a substructure opaque to the OS (e.g., archives and databases). Intercepting only file opens would force shards to be whole files, even though those files may be accessed quite sparsely. By intercepting more file APIs, or dropping down to the driver level, one might do somewhat better, but would still have to break files at arbitrary boundaries. Consider the case of two archives, differing from each other by a few constituent parts out of thousands. If the archives are broken into arbitrary pages, the redundancy would go undetected, whereas if they are broken at constituent boundaries, the management of the resulting shards would be far more optimal.

The only way to accomplish this is to exploit semantic knowledge about file structures that exists *above* the level of the OS. For example, although the OS does not understand the archive formats used by Java, the Java Runtime Environment (JRE) does understand them.

Consequently, PDS views the OS-level virtualizer as the *base virtualizer* (BV) within a hierarchy of virtualizers that can potentially operate at higher levels (see Figure 5). An asset can include a set of *non-base virtualizers* (NBVs) that intercept the APIs of subsystems such as the JRE. Deciding to implement an NBV for a particular subsystem is a pragmatic decision. PDS provides a general mechanism for adding NBVs and provides a JRE virtualizer, both to serve as an example, and because it is useful in its own right. This virtualizer ensures that shards correspond to the constituent parts of the various zip-format archives employed by Java and not to the archives as a whole.

The BV gives each NBV control when its subsystem is activated, assisting the NBV in intercepting additional APIs not intercepted by the BV itself. NBVs may interact directly with the deliverer to fetch shards. Otherwise, NBVs operate entirely within the sandbox provided by the BV, and hence they can use information in the VAI (virtual asset image).

In fact, the code of each NBV is itself added to the VAI of each asset during preparation, so that its version level is always correct for the asset it is servicing (Section 3.3.5 will elaborate on this issue). Digest-based shard storage ensures that the logical duplication implied by embedding the NBV code in every asset's VAI doesn't result in physical duplication.

Because each non-base virtualizer may have its own preparation requirements, the preparer is designed in a modular fashion with plug-ins corresponding to virtualizers that need prepare-time support.

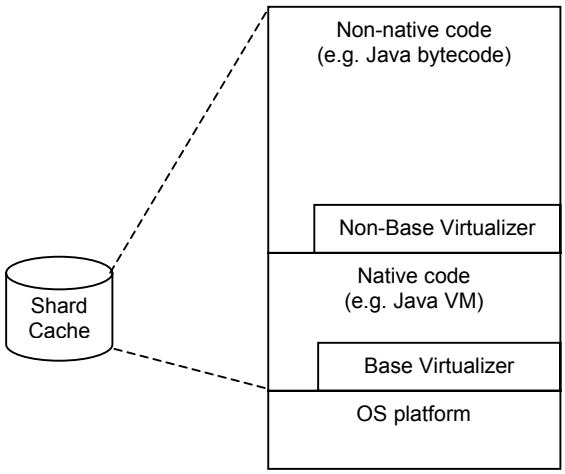

Figure 5 *Virtualized execution stack*

2.3.3 Base Virtualizer Details

The chief goal of the BV is to make an asset's post-install image visible to the asset without being visible to any other software running on the target machine. NBVs rely on this also, because code and data needed by NBVs are simply added to the VAI (virtual asset image) at prepare time. We have previously discussed how the BV filters API calls so as to handle only those that affect the VAI. We now provide more detail on what happens after a VAI-relevant API call is intercepted.

OS APIs are complex, so it is potentially error prone to interpose logic between applications and the OS. To minimize errors, we make it a goal to minimize the amount of logic interposed, observing the principle of "redirect, don't emulate." As much as possible, PDS observes a one-to-one correspondence between virtual API calls and real ones, with only the name of the resource (the file name, the registry key, etc.) being altered. For example, when a file open request is intercepted, PDS's response will be to open a shard in the cache, if the mode of opening prohibits modifications. If the mode of opening permits modifications, PDS makes a one-time copy of the file to another location and opens the copy.

Allowing shards to be opened directly in the cache improves efficiency, when it is possible to do so, and is a key to achieving low overhead. However, some assets modify their VAI during execution, and count on being able to do so persistently. Recall that the VAI is stored as a set of shards and that these shards may be in a shard repository that has been directly exposed to the executor. This repository may be read-only and/or shared across machines. Even if the shards are in a local writeable cache, they are potentially shared by multiple assets. Even within an asset, a single shard may represent multiple virtual entities that happen to have the same bit pattern (as shown in Figure 4). Thus, the BV cannot allow shards to be modified in place. Instead, it uses *shadow areas*, which are persistent stores in the local machine containing virtual entities that started out as shards but have since been modified by the asset. The file shadow area is a file tree on local disk, the registry shadow area is kept within the registry itself, and other forms of system metadata may require still other forms of shadow area.

The most challenging part of the PDS executor design concerns those cases where we are forced to deviate from the "redirect, don't emulate" principle. We consider three such cases next.

2.3.3.1 Separating Metadata from File Contents

As pointed out earlier, a modest problem arises from the fact that a single shard can represent many virtual resources. The contents of two virtual files that share the same bit pattern will be represented by the same shard. These files can have different names, creation dates, permission attributes, etc. Thus, APIs that involve the retrieval of metadata about files cannot just be redirected to a shard but must be emulated. PDS stores file metadata in separate metashards generated from the file trees of the VAI (virtual asset image) at prepare time. In fact, the shard containing the contents of the file does not have to be present in order to answer many queries about the file, and this can improve performance substantially, as will be seen in section 3.

The need to provide accurate metadata forces us to deviate from our desire to intercept only path directed requests, because some APIs permit the retrieval of metadata from open handles. Thus, PDS retains all handles that are open to shards in an efficient lookaside table. While it allows most handle directed requests to pass immediately to the OS, those that retrieve metadata must be intercepted and emulated. Handle-close operations are intercepted in order to remove handles from the table.

2.3.3.2 Providing Accurate Sharing Semantics

PDS's asset execution model supports multiple processes executing within the same asset. Such processes cannot be isolated from each other, but must be able to communicate through the VAI. For example, all file sharing and file locking capabilities must execute correctly between the processes of an asset, even though spurious sharing violations must be avoided between processes in different assets.

Adhering to the "redirect, don't emulate" principle, we could try to ensure that two handles open to the same virtual file are always open to the same real file, while two handles open to different virtual files are always open to different real files. Then, the OS would be responsible for all sharing semantics. Unfortunately, this solution, if adopted, would mean that no handles could ever be open to shards. If all shards must be copied to the shadow area in order to be opened, performance is seriously degraded.

Instead, we compromise as follows. Recall that each asset has its own file shadow area, where any shards modified by that asset are copied. PDS ensures that there is a one-to-one correspondence between virtual and real files within this area. This makes the OS responsible for all sharing semantics in the shadow area (and even for file metadata retrieval from this area, since there is no reason for PDS to be involved). However, as long as a virtual file is mapped to a shard, PDS will emulate the sharing semantics, just as for metadata retrieval. What makes this workable is that shards are copied to the shadow area, correct metadata assigned, and virtual operations shifted there, as soon as there is any potential for mutation. Once this shift has occurred, it is permanent for that virtual file of that asset. Thus, PDS is only required to provide correct read/read sharing, which is a much simpler problem than read/write and write/write sharing. It would even be a trivial problem except that Windows allows a reading process to exclude other readers. PDS's emulation

ensures that this exclusion operates only at the virtual file level and only within an asset.

There are some additional complexities. First, the act of shifting from the shard cache to the shadow area for a particular virtual file must be done transactionally. Two processes of the same asset that independently decide to make this shift must not collide, and a process that is opening a file for reading must be directed to the shadow area copy if it exists, even though this shift may just have taken place in a different process. This is readily accomplished using system-provided inter-process synchronization facilities.

A more subtle problem involves the status of processes that open files for reading but with concurrent writing allowed. It is impossible to determine whether this is being done so that concurrent writes can be observed, in which case the only safe course is to shift the virtual file to the shadow area immediately. Otherwise, another process could open the file for writing, shifting it to the shadow area and leaving the reading process's handle still open to the shard. The writing process would execute correctly, but the reading process would fail to see the changes.

If we assume, conservatively, that all cases of read/write sharing require the readers to observe the writes, we will copy many files unnecessarily, since standard libraries tend to allow writing by default when opening for reading. Instead, PDS assumes by default that a reader does not care about concurrent writes even if they are allowed. The file is thus opened to the shard in this case. If the virtual file is later shifted to the shadow area while the original handle is still open, we detect this potential safety problem and report the fact in a log. The asset can then be reprepared with information from the log made available. The preparer marks appropriate areas of the file tree for *strict sharing*, causing the more conservative algorithm to be used.

2.3.3.3 Dynamic Linking

Correct virtualization of the dynamic linking capabilities of the OS also requires work beyond merely redirecting file APIs. This happens because aspects of this linking are accomplished *implicitly* without any visible call to a system API. In Windows, executable images and libraries have *import sections* that refer to the *export sections* of other libraries (other OS's have similar facilities). So, lower level intercepts are needed to correctly ensure that imports are satisfied from the VAI instead of the real local file system when appropriate. Fortunately, Windows provides the ability to load executables and libraries without resolving their imports and provides enough public information to allow some of this resolution to be performed manually. So, PDS is able to analyze the imports and determine, for each one, the correct module to load, either from the VAI or from outside the VAI.

Once the correct module to load has been identified, PDS can use normal system APIs to load the module under the usual "redirect, don't emulate" principle. But, subsequent APIs that query the names of loaded modules must be intercepted to maintain the virtualization (since the actual module loaded may be a shard in the shard cache, with an arbitrary name).

Another noteworthy aspect of dynamic linking is its use of *search paths* to resolve the actual identity of the module to be loaded. In a PDS asset, the virtual search path may include directories within the VAI that don't actually exist in the local machine. Thus, the search itself must also be emulated.

2.3.4 Bootstrapping

Recall that a PDS asset contains the correctly resolved closure of its dependencies, except dependencies on the OS. But, correct execution of the asset will only have been verified for particular versions of PDS itself. An important secondary goal, therefore, is to reuse PDS's virtualization technology to ensure that the correct version of each PDS component is used with each asset that is executed. We've already discussed how any NBVs (non-base virtualizers) used by the asset become part of the asset, which solves the problem for NBVs. But, the BV (base virtualizer) and the deliverer are also part of PDS and can affect the execution of the asset.

Making the BV and the deliverer be part of the asset would be, in some sense, circular, but we are able to get the equivalent by making these two low-level components into *microload* assets. A microload asset is an asset with the following two characteristics. (1) Its logic is so simple that it doesn't need the BV to execute correctly. (2) When executed, it creates a directory (its *microload directory*) named for its own asset id and stores some number of files there (typically, code libraries). The microload assets for each version of the deliverer and the BV simply make microload directories containing those components. Because all assets have unique ids, every distinct version of these components will have a different id and hence a different directory.

The PDS bootstrap is a tiny program that knows how to load a deliverer from its microload directory and pass a launch document to its startup function. Thus, the bootstrap makes almost no assumptions about the rest of PDS, making changes to the bootstrap itself into very rare events. The bare minimum installation of PDS consists of this bootstrap plus one microload directory containing a version of the deliverer. In the current prototype system, the bootstrap and microload pieces are about 0.5 Mbytes is size.

The startup function of every deliverer extracts from the launch document the asset id of the target asset plus the asset ids of the microload assets for the required deliverer and BV. Assuming only that this deliverer knows how to locate the microload asset for the desired deliverer, it can execute a chain of microload asset executions followed by the target asset execution that will create the correct configuration.

2.3.5 Executor Design Summary

Interception at the level of the OS API boundary does introduce complexities and vulnerabilities that don't exist at the hardware level. However, PDS benefits from its own design goal of selective virtualization, which bounds the portion of the API that must be intercepted. By steadfastly insisting on making the OS do as much of the work as possible, we are able to limit PDS's actual OS emulation to a few key areas discussed in the previous subsections. Although we cannot prove that these exhaust the set of issues that might arise, it is encouraging that PDS has been able to provide solid virtualizations of a number of useful assets already, with problems and failures being fewer and farther between.

3. MEASUREMENTS

We first present some measurements of PDS's execution time overhead, which gives a sense of what it costs to execute software under PDS even after all of the shards are present locally. We then present measurements of the degree to which PDS's use of shards is able to reduce working set size and detect and eliminate the redundant storage and transfer of bits when deploying software progressively.

3.1 Execution Time Overhead

We measured execution time overhead with two tests. First, we ran the Apache web server [1] both under PDS (with the shard repository on a local file system) and natively. We measured performance using the standard benchmark distributed with apache. The benchmark starts multiple clients (we used 20 in our measurements), each of which repeatedly retrieves a web page from the server. In our test, the clients did not run under PDS, only the server did, and all socket connections were local. We found that this test ran very slightly *faster* under PDS than native (8.4 ms per request as opposed to 8.7). The reasons for this counter-intuitive result are discussed below.

In a second test we measured the time to start up Eclipse 3.0 [6], with an empty workspace, both under PDS (with a local shard repository) and natively. The startup time averaged 12.6 seconds in native mode, and 12.9 seconds under PDS, for an overhead of 2.4%.

The fact that PDS exhibits slightly negative overhead in one test and negligible overhead in another can be explained by the results of some microbenchmarks which we also ran.

The first microbenchmark paired the **CreateFile** and **CloseHandle** calls, executing the pair repeatedly on a file (opening the file for reading, then closing it). When this benchmark is run natively, it takes 375.9 microseconds per iteration. When the same benchmark is run under PDS, but with the file residing outside the VAI (virtual asset image), it takes 424.9 microseconds (13% overhead). When the file is inside the VAI, PDS adds 56% overhead for a timing of 587.4 microseconds per iteration.

The second microbenchmark paired the **FindFirstFile** and **FindClose** functions, executing the pair repeatedly on a file. This API is used very heavily by windows applications (due to its heavy use by the C runtime library) to test file existence and retrieve metadata. When this benchmark is run natively, it takes 124.4 microseconds. When run under PDS with a file outside the VAI, it takes 168.4 microseconds (an overhead of 35%, but smaller in absolute terms than the overhead added to **CreateFile** in the same circumstances). But, when the file is inside the VAI, PDS executes each benchmark iteration in only 66.1 microseconds, for a *saving* of 47%! Recall that, in section 2.3.3.1, we listed metadata retrieval as a case where we emulated the function rather than delegating to the OS. Because PDS packs metadata efficiently into metashards which it then memory-maps, it is apparently able to deliver the information substantially faster than the OS can.

In both real benchmarks, the files resided inside the VAI, and so they incurred a 56% overhead for **CreateFile** but a 47% saving for **FindFirstFile**. Since the latter API is one of the most heavily used ones in Windows, the average overhead of PDS was near-zero. We cannot claim that this will always be the case. But, by accepting some overhead on less frequently used APIs and making up the difference on others it is possible, with this approach, to have a very low-cost virtualization.

3.2 Working Set versus Asset Image Size

A primary reason for PDS adopting the shard strategy was to help in managing working sets. To measure the effectiveness of our solution, we defined a test asset consisting of Eclipse 3.0 and IBM's version of Java 1.4.2 (this is the asset designated as E3 J2 in Table 1 of the next section). The asset's original image was 149 megabytes in size, and its effective size in the repository (see next section) was 106 megabytes. But, after starting the asset on a machine with an empty shard cache, the asset was able to start up after transferring only 22 megabytes to the shard cache.

Achieving this working set required the use of the JRE virtualizer. An alternative preparation of the asset which did not break up archives for use by the JRE Virtualizer required the transfer of 72 megabytes to the shard cache.

3.3 Redundancy Elimination

Measurements of redundancy employed a PDS shard repository containing 12 assets. These were constructed using three different versions of Eclipse (2.1.2, 2.1.3, and 3.0.1, designated as E1, E2, and E3, respectively) and three different versions of Apache Tomcat (4.1.29, 4.1.30, and 4.1.31, designated as T1, T2, and T3, respectively). Eclipse is dependent on a Java Runtime, and Tomcat actually requires a full Java Development Kit including the compiler. These dependencies were satisfied by pairing the original six products with two different versions of IBM's Java product (1.4.1 and 1.4.2, designated as J1 and J2, respectively) to form complete assets. All assets were prepared for the JRE Virtualizer as well as the BV.

Table 1 gives the measurements for these assets. All sizes are in megabytes rounded to the nearest megabyte.

The *image size* column contains the size of each asset image after installation and before preparation (the sum of the sizes of the file trees containing both the Java version and Eclipse or Tomcat version employed). The *raw prepared size* is the size that the asset would have in the shard repository if no attempt was made to recognize redundant shards.

The act of preparation initially increases the size of the asset. This happens almost entirely due to the special preparation done to support the JRE Virtualizer. Specifically, archives are broken up into individual members, which are uncompressed in the process. Then, the same member may be counted multiple times due to its appearance on multiple classpaths. Finally, the original archive is also left as a shard in its own right, since it may be manipulated by non-Java code. We consider this initial growth acceptable, since it is in service of a smaller working set size, as discussed in the previous section.

The raw prepared size is an artificial number, since PDS always detects redundant shards and stores them only once. The *actual prepared size* shows the effects of this feature. This is the size that each asset would have if stored in a shard repository by itself.

The *effective prepared size* column gives the effective size of each asset when stored in a shard repository along with the others. The number was computed by counting, as part of each asset, those shards that were unique to the asset, then amortizing across all

Asset	Image Size	Raw Prepared Size *(no redundancy removal)*	Actual Prepared Size *(after redundancy removal)*	Effective Prepared Size *(when stored in a common repository)*
E1 J1	110	356	211	66
E1 J2	115	372	225	68
E2 J1	110	356	212	66
E2 J2	115	373	225	68
E3 J1	144	280	268	103
E3 J2	149	296	281	106
T1 J1	88	224	156	38
T1 J2	93	248	162	38
T2 J1	89	227	156	38
T2 J2	93	249	163	38
T3 J1	89	228	157	41
T3 J2	94	250	164	41
Total	1279	3439	2380	711

Table 1 *Asset Sizes. E1, E2, E3 represent different versions of Eclipse. T1, T2, T3 represent different versions of Tomcat.*

affected assets the sizes of those shards that are shared. Notice that this represents a considerable savings over the actual prepared size, and more than compensates for the increase due to JRE preparation, resulting in an effective asset size smaller than the original asset image. This effect only increases as the number of highly similar assets is aggregated. We consider the example shown here to be realistic, since it includes some very similar assets (the three Tomcats), some only moderately similar assets (the two Javas and the three Eclipses), and coarse-grained component sharing (use of the same Java version by different assets).

The significance of this measurement is not only, or even primarily, in the reduction of footprint in shard repositories or shard caches. Rather, the times taken to transmit assets over a network are substantially reduced when some of the shards are already found in the local cache. This can be particularly useful when updating from one version of an asset to a newer one.

4. RELATED WORK

The use of virtualization as a software abstraction of the underlying hardware machine was developed by IBM in the 1960s [28]. A spectrum of Virtual Machines of different sorts are in use today, ranging from runtime environments for high-level languages like Java [15] and Smalltalk [9] to hardware-level virtual machine monitors (VMMs) such as VMware [30] and Xen [4].

The level of indirection provided by the virtual machine layer enables the software running above it to be decoupled from the system beneath it. This decoupling enables the virtual machine layer to control or enhance the software running above it.

High-level languages use their runtime environments both to enhance the functionality of underlying hardware and OS and to achieve portability across hardware and OS implementations. While PDS does not have these aims, it does use virtualization to mask the idiosyncrasies that arise within an operating system instance as individual machines are configured differently.

Virtual machine monitors like VMware exploit the decoupling to fully isolate the software stack running above it from the host environment, thus enabling sandboxed environments for testing, archival and security. The CMU/Intel work on Internet Suspend/Resume [13][14] and the Stanford Collective project [22][23] use the ability of a VMM to easily capture both the persistent and volatile state of a sandboxed environment to enable mobility of end-user environments over a network. Virtual Appliances (also part of the Stanford Collective project) exploit the VMware VMM for simplifying the deployment and maintenance of software environments [22]. A key difference between these approaches and ours is that PDS implements a weaker form of decoupling than a traditional VMM, by isolating only the non-OS dependencies of the asset from the host environment. While this does not provide the full isolation sandbox that a VMM does, it enables separately deployed applications to co-exist and interact as peers in the same host environment without the risk of conflicts (Figure 1). This allows PDS to be used in scenarios where separate vendors deploy different parts of a complex environment.

Utilities like Debian *apt* [8] simplify the maintenance of software packages, but do not provide isolation in the sense of enabling conflicting versions of a component to co-exist in the same (virtual) namespace.

Managed container frameworks like J2EE and .NET provide network deployment and management features, but they are language specific, and require the use of framework APIs. Other language-specific solutions for software deployment and maintenance are Java Web Start [11] and OSGi [19].

Zap [20] is an implementation of a virtualization layer between the operating system and the software. The goal of Zap is migration of process groups across machines, not software deployment and serviceability.

A number of recent startups like AppStream [2], Endeavors [7], Softricity[25] use file-system based approaches to provide centrally managed software deployment and maintenance solutions for Windows desktops. Desktop applications are generally self-contained applications, whose non-OS dependencies are easily be bundled within a single file system mount point, or self-contained directory.

5. FUTURE WORK

Although PDS's observed aggregate runtime overhead is usually low in practice, it was seen in section 3.1 that this is a consequence of running faster-than-native for some APIs, and hence it may not always be the case. Microbenchmarks reported in that section indicate that other APIs still have significant overheads. Performance tuning has not been a priority until now, but we expect to put significant effort into reducing overheads further, especially when the asset is operating on files outside the VAI.

The deliverer in the PDS prototype uses either servlets or off-the-shelf file system software when demand-fetching shards from a

remote location. Thus, we see high per-shard latencies when the shard repository is remote. We intend to experiment both with better-performing servers and with the exploitation of network file serving technologies as we alluded to in section 2.2. The highly preliminary nature of our deliverer prevented us from reporting network startup times in this paper, but we expect to present such results in a future paper. Interestingly, despite suboptimal network performance in the prototype, experimental users of PDS still consider startup to be "fast", because it is such a relief not to have to go through an installation step.

The JRE Virtualizer significantly reduces the amount of data required to startup an asset (in some versions of Eclipse by as much as 60%). However, this savings is achieved at the cost of introducing many more (smaller) shards. Currently, the latency overhead of retrieving these shards overbalances the bandwidth savings. After the deliverer is redesigned for lower per-shard latency, we expect to measure a positive gain from the use of hierarchical virtualization. Until then, our claims of the benefits of this approach should be taken as preliminary.

Even if demand-fetching of individual shards remains a high latency operation, we expect to exploit the fact that assets typically require shards (classes and files) in bursts. Efficient prefetching based on previously observed sequences has been claimed by others in the network deployment business (e.g., [2]). We already have an experimental predictive prefetching system working, although it has so far been used only to optimize the transfer of the initial working set at asset startup, and we were unable to collect data from it for this paper. Future research will focus on how prefetching can be optimized in a context that also includes virtualization (where information from the virtualizer can be used to tune and control the prefetch).

To support hierarchical virtualization, the BV provides a general "interception assistance" mechanism for use by NBVs. We are now considering whether this mechanism can be exploited for other purposes. For example, could we utilize this technology to improve the serviceability of an asset by introducing probes and fixes dynamically? Could it be used to provide a transparent licensing mechanism for applications (one that could be introduced without recoding those applications)? We are pursing each of these thoughts in collaboration with other research groups in our organization.

Currently, setting up to prepare an asset is a manual process. A failure to properly inventory the asset image leads to errors that can only be found by executing the asset. We are exploring tools to solve these problems. For example, traditional VMMs can be used to simplify the collection of information about how installations change the system, thus automating the asset image inventory. VMMs can also be used to at least speed up the cycle of prepare, then test, then re-prepare which is likely to be needed in practice no matter how well we automate things.

Our current preparer assumes it knows nothing about how the asset image is used during execution: every part of it is assumed to be equally important. Static analysis and dynamic trace feedback techniques can be used to further optimize preparation. We already use a primitive form of dynamic trace feedback to automate preparation for the JRE virtualizer, but much more can be done in this area.

Finally, efficient exploitation of the shard concept depends on our ability to identify semantically meaningful units within larger entities. We have developed solutions for one of the most obvious cases (zip-format archives used by the JRE). But we don't know what other examples of this phenomenon may prove to have equal or greater pay-off in the future.

6. CONCLUSION

PDS is a novel solution to the problem of deploying and managing complex software stacks. By treating assets as immutable and with their own view of their virtual file spaces, along with the ability to share components between assets, PDS allows multiple assets to simultaneously execute on the same machine. The automated redundancy removal introduced through cryptographic hashes allows the efficient delivery of many assets which share common sub-components.

Furthermore, with the exception of a small bootstrap code (about 0.5 Mbytes in the current prototype), PDS's own virtualizers are embedded in every asset. The shard design ensures that the duplication implied by this is avoided in the physical shard storage. This embedding allows assets to be unaffected by subsequent PDS virtualizer evolution, further enhancing the ability to service and support deployed assets in the field.

There are at least two benefits that result from such a model. First, the end-user's perceived complexity of the deployed environment is lowered, because its internal structure is hidden from the user. Second, it enhances the serviceability of deployed environments, because every asset represents an immutable state of some installed image, and no user can have an image that is in-between two supported asset versions.

PDS uses a selective and hierarchical approach to process-level virtualization, which enables multiple assets to co-exist and interact as peers in the host machine environment, without incurring a significant performance penalty. This enables multiple vendors to deploy different parts of a complex commercial environment, which would be difficult to accomplish with a full isolation sandbox approach based on a virtual machine monitor. On the other hand, PDS cannot isolate environments at an OS level the way that virtual machine monitors can. Thus, the two approaches are fundamentally complementary to one another. In fact, the two approaches could be used together to get the benefits of both.

7. ACKNOWLEDGMENTS

We thank Frank Cavallito and Jobi George for making significant contributions to the implementation of the PDS prototype system, and to Nick Mitchell, Harold Ossher, Gary Sevitsky and the referees for their comments and suggestions on this paper. We are also grateful to Alfred Spector and the management at IBM Research for their enthusiastic support of this project.

8. REFERENCES

[1] Apache open-source web server. http://www.apache.org.

[2] AppStream Inc. http://www.appstream.com.

[3] Arthorne, J., Laffra, C. Eclipse 3.0 FAQs. *Addison-Wesley* 2004.

[4] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A. XEN and the Art of Virtualization. *In Proceedings of the 19th ACM Symposium on Operating System Principles*, Oct 2003.

[5] Brittain, J. Darwin, I. Tomcat: The Definitive Guide. *O'Reilly*. June 2003.

[6] Eclipse Open, Extensible IDE. http://www.eclipse.org.

[7] Endeavors Inc. http://www.endeavors.com.

[8] Debian open-source OS. http://www.debian.org.

[9] Goldberg, A., Robson, D. Smalltalk-80: the language and its implementation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1983.

[10] DuBois, P. MySQL (2nd edition). *Sams press.* March 2005.

[11] Java Web Start. http://java.sun.com/products/javawebstart/.

[12] The JBoss Group. JBoss 4.0 – The Official Guide. *Sams press*. April 2005.

[13] Kozuch, M.A., Helfrich, C. J., Hallaron, D.O., Satyanarayanan, M. Enterprise Client Management with Internet Suspend/Resume. *Intel Technology Journal*, *Vol 8, Issue 4,* Nov 2004.

[14] Kozuch, M. A., Satyanarayanan, M. Internet Suspend/Resume. *In* Proceedings of the *4th IEEE Workshop on Mobile Computing Systems and Applications, NY,* June 2002.

[15] Lindholm, T., Yellin, F. The Java virtual machine specification, 2nd Ed. Addison-Wesley, Reading, MA, 2000.

[16] Lotus Workplace Client. IBM Software Group. http://www.lotus.com/products/product5.nsf/wdocs/workplaceclienttech.

[17] Lowell, D.E., Saito, Y.Samberg, E.J. Devirtualizable Virtual Machines Enabling General, Single-Node, Online Maintenance. *In Poceedings of the 11th ASPLOS,* Oct 2004.

[18] Open Office suite. http://www.openoffice.org.

[19] OSGi specification. http://www.osgi.org

[20] Osman, S., Subhraveti, D., Su, G., Nieh, J. The Design and Implementation of Zap: A System for Migrating Computing Environments. *ACM SIGOPS Operating System Review, Vol 36, Issue SI*, Dec 2002.

[21] Rosenblum, M. The Reincarnation of Virtual Machines. *QUEUE Vol 2, Issue 5*, July 2004.

[22] Sapuntzakis, C., Brumley, D., Chandra, R., Zeldovich, N., Chow, J., Lam, M.S., Rosenblum, M. Virtual Appliances for Deploying and Maintaining Software. *In Proceedings of the 17th Large Installation System Administration Conference*, Oct 2003.

[23] Sapuntzakis, C., Chandra, R., Pfaff, B., Chow, J., Lam, M.S., Rosenblum, M. Optimizing the Migration of Virtual Computers. *ACM SIGOPS Operating Systems Review*. Dec 2002.

[24] Secure Hash Standard. FIPS publication 180-2, National Institute of Standards and Technology.

[25] Softricity Inc. http://www.softricity.com.

[26] Sugerman, J., Venkitachalam, G., Lim, B-H. Virtualizing I/O Devices on VMWare Workstations's Hosted Virtual Machine Monitor. *In Proceedings of the USENIX Annual Technical Conference, Boston*. June 2001.

[27] Thain, D., Livny, M. Parrot: Transparent User-Level Middleware for Data Intensive Computing. *In Proceedings of the Workshop on Adaptive Grid Middleware*, 2003.

[28] The IBM Mainframe, history and timeline. http://www-1.ibm.com/servers/eserver/zseries/timeline/.

[29] Tomcat open-source servlet engine. http://jakarta.apache.org/tomcat/.

[30] VMWare Inc. VMWare ACE. http://www.vmware.com.

[31] WebSphere Studio Application Developer. IBM Software Group. http://www-306.ibm.com/software/awdtools/studioappdev/.

[32] Whitaker, A., Shaw, M., Gribble, S.D. Scale and Performance in the Denali Isolation Kernel. *In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston*. Dec 2002.