Morpheus: Learning Configurations by Example

Deepak Jeswani, Rahul Balani, Akshat Verma, Kamal Bhattacharya
IBM Research, New Delhi, India.
Email: {djeswani,rabalani,akshatverma,kambhatt}@in.ibm.com

***Abstract*—Cloud computing presents a model to automatically deploy workloads using standard templates. However, reconfiguring the application to work in the new environment is manual and time-consuming. In this work, we present *Morpheus* to automatically configure new instances of a workload by leveraging a pre-configured instance. *Morpheus* uses a configured instance of a workload, annotated with the configuration information, and automatically configures new instances of the workload. It is important to note that it can configure application instances that may be installed differently or may be running a different version of the application, middleware or operating system.**

***Morpheus* first maps configuration files from the template instance to configuration files in each new instance. In the second phase, configuration nodes in source file are mapped to the target file and configuration information is automatically added in the target instance. The mapping is performed using text and structure analysis in the configuration files. We used *Morpheus* to automatically deploy new workloads for a wide variety of multi-tier enterprise/web applications. It was able to deploy them without any manual intervention in 13–68 minutes for 100% of the workloads.**

I. Introduction

Cloud computing presents a fundamental new computing paradigm allowing end users to request resources and run their applications in real time. Application implementation in Cloud computing environments consists of application implementation and configuration. IaaS clouds (e.g., Amazon EC2 [1]) simplify application implementation by allowing virtual machines with a configured operating system to be provisioned in the matter of minutes. Virtual application templates (e.g., [17]) extend the on demand application delivery model even further by allowing entire application software stack to be provisioned on the fly. However, application (re)configuration is still necessary and every instance of an application needs to be configured *manually* incurring significant labour cost.

Category	WebLogic	Websphere	Jboss
Hardware	0.027944076	0.024309286	0.141709967
Software	0.377245027	0.444167473	0
Installation	0.007513603	0.009850817	0.051487955
Configuration	0.020336581	0.02339109	0.447803496
Integration	0.524726036	0.452152726	0
Test/Delploy	0.042234677	0.046128607	0.358998583

TABLE I
BREAKUP OF IMPLEMENTATION COSTS (NORMALIZED) FOR WEB SERVERS. SOURCE ([2], [3])

Simple gateway-based applications benefit from the cloud model as their configuration is usually very simple. On the other hand, configuration of enterprise application stack is usually very complex. We study the overall implementation costs for popular web servers in Table. I. We observe that hardware contributes less than 15% of total costs for all web servers. Further, installation costs of enterprise middleware is also very small (less than 5%). One of the significant component of overall costs is the configuration, integration, and deployment costs contributing more than 50% of total costs. Combined with the fact that a typical cloud or datacenter runs multiple instance of the same application (eg., in Test&Dev, Staging and Production) with minor changes, application configuration costs are added for each distinct instance. Further, if applications migrate between a data center and a cloud (as in a hybrid cloud model [14] or between clouds (inter-cloud [16]), the application configuration cost is incurred for every migration.

Our Contributions: In an earlier work, we had created the PoVMiner system [6] to discover and annotate environment, performance and operation related configuration parameters, which are collectively known as Points of Variability (PoV), in an application instance. In this work, we present the design and implementation of *Morpheus*, a system to automatically configure multiple instances of the same application. However it should be noted that current scope of *Morpheus* is limited to configuration and doesn't handle installation issues like missing components.

For reconfiguring the application, *Morpheus* uses a sample configured application instance, which is annotated with the Points-of-Variability (PoV), to automatically configure any other application instances in the same or different environment. *Morpheus* can configure application instances that may be installed differently or even be running a different version of the application, middleware or operating system. It uses the concept of *linkage markers* to map parameters in configuration files across application instances. We combine the content in configuration files with the structure (or layout of content) in the files to uniquely identify PoVs in configuration files for new instances. *Morpheus* can also deal with unconfigured application instances where PoVs may be missing by identifying the location in the configuration file, where PoVs need to be added. We evaluate *Morpheus* on a wide variety of workloads and observe that it is very effective in automatically configuring new application instances using the example instance.

II. Background and Problem Formulation

In this section, we present a formal description of the problem and some motivating scenarios.

A. Problem Description

We consider the problem of automatically configuring (target) application instances A_T from one example (source) configured instance A_S. Each application consists of a set of images. Each configuration parameter that varies across environments is captured as a Point-of-Variability (PoV) parameter. Typical examples of environment-related PoVs include IP addresses, MAC addresses, ports, hostname, usernames, passwords and authentication tokens. Configuring an application to work in an environment requires setting the

978-3-901882-50-0 ©2013 IFIP

PoVs in configuration files to the correct value dictated by the environment. We use I_S^i to denote image i of the example application instance A_S and I_T^i to denote an image of a target application instance A_T that we want to configure. The input to *Morpheus* is the set of configuration files in each source image of the application instance A_S. For each configuration file $F_S^{i,j}$, we are provided with a set of $< PoV, Location >$ tuples, which indicate the location of each PoV in the file.

The goal of *Morpheus* is to annotate PoVs for each configuration file $F_T^{i,j}$ in the target application that corresponds to some configuration file $F_S^{i,j}$ in A_S. Once the PoVs are annotated, they will be replaced by the actual values of the PoVs in the target environment. If the PoVs are not found in the target environment, *Morpheus* annotates locations in configuration files, where the PoVs need to be added along with the exact format to represent the PoV. Hence, *Morpheus* needs to solve two problems

1) For each source configuration file $F_S^{i,j}$, find its equivalent target file $F_T^{i,j}$
2) For each $< PoV, Location >$ tuple in $F_S^{i,j}$, find either (a) the tuple in $F_T^{i,j}$ OR (b) find the location in $F_T^{i,j}$ to insert the PoV.

B. Motivating Scenarios

We have developed *Morpheus* in the context of two cloud use cases.

1) **Application Migration**: Migration of enterprise applications to cloud is manual and expensive. Often applications are migrated using lift-and-shift of VM images from one environment to another. However, applications running on older platforms are often reinstalled during migration to take advantage of the new environment. Migration in such scenarios consists of (a) discovery of the source environment (b) reinstall of application stack in the target environment and (c) reconfiguration of the application in the target environment. Various tools exist for discovery [12] and installation [5]. However, application reconfiguration is time-consuming and manual. Since operating system, middleware or application versions may change during migration, the complexity of reconfiguration is compounded even further. *Morpheus* simplifies the migration process by using the source configuration to automatically configure the target application instance.
2) **Rapid Application Deployment**: One of the key features of cloud is rapid provisioning of virtual machines. Virtual application templates [17] extend the abstraction of provisioning from virtual machines to applications by allowing multiple virtual machines, which together constitute an application, to be deployed automatically. However, configuring the provisioned application to work seamlessly in the deployed environment is outside the scope of the provisioning process. A system like *Morpheus* complements the application templatization process allowing deployed target application instances to be fully configured by leveraging a set of configured golden master images, which constitute the application.

C. Input: Configuration Annotation

As mentioned earlier, the set of configuration files from application instance A_S need to be annotated with location of PoVs before they can be used by *Morpheus* as input. This preprocessing step can be done either manually, by an application expert, or automatically through semantic analysis of the images based on values of environment variables, such as in PoVMiner [6]. Although time-consuming and expensive, the manual effort is precise and only needs to be undertaken once for A_S. On the contrary, the current automated solution is cheaper and faster, but often slightly less accurate.

PoVMiner automatically identifies configuration files in an instance and represents them as AST's (Figure 5). Their subsequent analysis is based on the observations that (1) configuration files typically consist of key-value pairs which define the PoVs and (2) the keys, along with other contextual hints, indicate the semantics associated with the corresponding value. The output of PoVMiner, which consists of AST nodes corresponding to the PoVs in each configuration file, is finally serialized and provided as input to *Morpheus*.

D. Challenges

Automatically configuring an application presents significant challenges even when a sample application instance is available. As we note before, configuration requires (a) identifying configuration files in the target image and (b) identifying PoVs in the target configuration files. It is important to note that the target application instance may contain a different version of the application, middleware or the OS, may be installed with entirely different installation options, and the filesystem structure of the target image may vary widely from the example source images. This presents significant challenges in automatic reconfiguration. An example is shown in Figure 1 to highlight the challenges described below:

- **Change in configuration file location and name**: Different images may have entirely different filesystem structure and hence the absolute path of the application may change across instances. Further, applications may be installed by different user accounts and may specify installation directories, which may lead to very different location of the configuration files. Further, change of application, middleware or operating system version may lead to change in the application directory structure as well as file name (e.g., version number may be incorporated in the path).
- **Permutation of content**: Configuration files typically consist of a set of sections to capture various types of configuration. Keywords are used to demarcate sections and the sections can be re-arranged within a configuration file without the application getting impacted. Hence, the location of a PoV in a source file is not useful in identifying the PoV in a target file.
- **Missing content**: Configuration files may get populated during application configuration and therefore unconfigured applications may have entire sections missing.
- **Lack of Linkages**: One of the most significant challenge is the lack of linkages between elements in the source and target configuration files. We have already noted that location does not link PoVs across source and target due to re-arrangement. The actual values of the PoVs also differ between the source and target files (e.g., IP address is different for the source and target instances). Hence, it is extremely challenging to link elements in source files to elements in target files.

```
names.message_pool_start_size = 10
names.preferred_servers = (address_list =
      (address=(protocol=ipc)(key=n23))
      (address=(protocol=tcp)(host=nineva)(port=1486))
      (address=(protocol=tcp)(host=cicada)(port=1383))
           )

namesctl.trace_directory = /oracle/network/trace
namesctl.server_password = mangler
namesctl.internal_encrypt_password = false

sqlnet.identix_fingerprint_database=ofm
sqlnet.identix_fingerprint_database_user=<username>
sqlnet.identix_fingerprint_database_password=<password>

oss.source.encrypted_private_key=(source=(method=oracle)
                  (method_data=(username=andre_security_service)
                               (password=andre_security_service)
                               (sqlnet_address=andreoss))
                   )
oss.source.certificates=(source=(method=oracle)
                  (method_data=(username=scott_security_service)
                               (password=ascott_security_service)
                               (sqlnet_address=andreoss))
                  )
```

Source file: $ORACLE_HOME/network/admin/sqlnet.ora

(a)

```
oss.source.certificates=(source=(method=oracle)
                  (method_data=(username=user_secure)
                               (password=passw0rd)
                               (sqlnet_address=hostoss))
                  )
oss.source.encrypted_private_key=(source=(method=oracle)
                  (method_data=(username=user_secure)
                               (password=passw0rd)
                               (sqlnet_address=hostoss))
                   )

namesctl.trace_directory = /root/oracle/network/trace
namesctl.server_password = passw0rd
namesctl.internal_encrypt_password = false

names.message_pool_start_size = 10
names.preferred_servers = (address_list =
      (address=(protocol=ipc)(key=n23))
      (address=(protocol=tcp)(host=defaultserver)(port=8000))
      (address=(protocol=tcp)(host=bakupserver)(port=8090))
                  )

sqlnet.identix_fingerprint_database=odb
<-------------- Missing PoVs ------------->
```

Target file: $ORACLE_HOME/network/admin/sqlnet.ora

(b)

Fig. 1. Snippets of Oracle8 (SQL*Net) configuration file from source and target application instances where Oracle8 is installed in different directories referenced by $ORACLE_HOME. In addition, the PoVs in target file are unconfigured or absent.

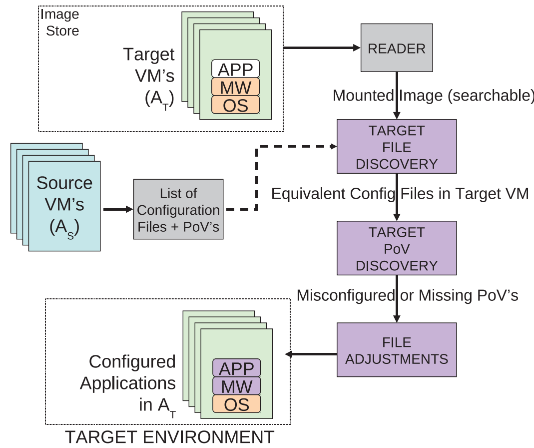

Fig. 2. *Morpheus*: Design Overview

III. Design

In this section, we describe the design of *Morpheus*, as illustrated in Figure 2, highlighting various techniques employed to solve the above challenges.

A. Morpheus *Configuration Framework*

Morpheus reconfigures target virtual machine in an offline manner. It does not require the application to be up and running or the virtual machines to be booted up for configuring the application. Reconfiguration using *Morpheus* is performed through a 4-step process.

- Reader: A reader module mounts the target virtual machine I_T^i and allows *Morpheus* to crawl the file system, identify any configuration related information, and make relevant changes.
- Target File Discovery: The *Target File Discovery* (TFD) module takes source configuration files as input. It scans the target image and identifies for each source file $F_S^{i,j}$, its equivalent configuration file $F_T^{i,j}$ in the image I_T^i.
- Target PoV Discovery: The *Target PoV Discovery* (TPoD) module searches through the identified configuration files $F_T^{i,j}$ to locate mis-configured or missing PoV's using the structure and context derived from annotations of the source configuration files. The PoVs are then annotated in the target file, if found or the location is identified to insert PoVs.
- File Adjustments: The *File Adjustments* module receives as input $< PoV, Location >$ tuples for each PoV found in the target files and a list of missing PoVs. It incorporates user feedback at this stage to resolve any ambiguities in the output of TPoD module. Mis-configured PoV's are replaced with their correct values, while the missing PoV's are added at the correct locations using the desired values and format extracted from the source configuration files.

B. Key Ideas and Approach

Morpheus uses the following key ideas to address the challenges in automatic application configuration of target instances.

- **Three-dimensional target file identification**: Our first idea addresses the problem of identifying target files, while dealing with change in location, name and content. We essentially identify target files by using similarity in (a) file location (b) file name and (c) keyword distribution in file content. Further, for file location, we use the insight that changes due to installation options are more likely in lower level directories. Hence, we give a higher weight to matches in lower level directories by using suffix match.
- **Hierarchical Element Mapping**: Lack of linkages and permutation of content imply that the source tuples can not be directly mapped to target file at all. To address this problem, we design a hierarchical element mapping scheme. In this scheme, a tree is created for the entire

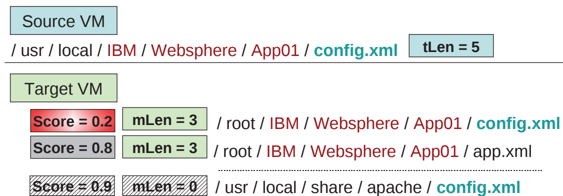

Fig. 3. Target File Discovery: An example of *Location-Name* filter. Last file in target VM is not considered because of the presence of first two files, but is shown here for explanation only.

configuration file (both source and target) and we try to map relevant nodes in the source file to respective nodes in the target file.

- **Fuzzy content markers**: Often, the key content in the source (sample configured system) and the target system is not identical. The reason for difference in content can be noise due to version change, timestamps inserted by application during install, etc. We thus map the source nodes to the target nodes in a probabilistic manner using structural information and content similarity of nodes (tags and attributes). This fuzzy logic is discussed in detail in section III-D2.
- **Leverage logical structure of configuration file**: Our most important idea is to use structure of the tree to map nodes that can not be mapped by content alone. As we note before, content (or value) of PoVs almost always differs between source and target files. However, a keyword in the same section as the PoV may match across the two files using the content marker. For instance, in the source file shown in Figure 1, the *username* and *password* keywords corresponding to PoV value *andre_security_service* under *oss* section match with multiple occurrences of identical keywords in the target file. In such cases, we use the logical structure (sibling in the same section) in conjunction with the keyword match to also match the PoV nodes.

C. Target File Discovery

The TFD module crawls a mounted image I_T^i to discover text (ASCII) files that may be potential target candidates for the set of input source files $\{F_S^{i,j}\}$. For each source file in the input set, it filters the set of candidate files through sequential application of two filters, *Location-Name* and *Key-Distribution*.

1) Location-Name Filter: The Location-Name filter tries to identify equivalent target files by exploiting similarity in file names and paths in a prioritized fashion. When full path and file name match fails, the underlying algorithm searches for files with identical names and the longest common suffix in file path as shown in Figure 3. For each non-zero suffix match with a target file, it records *mLen*, which is the length of the common suffix. In addition, to account for a change or swap in file names, it also records *mLen* for the target text files with non-zero suffix but non-identical names.

When all of the above checks fail to determine a set of candidate target files for a source file, the module stores and records all the files with identical names but zero common suffix in their file paths. The latter case is typically encountered in the case of a missing file or a version upgrade for an application that incorporates its version number in naming its last-level directory for storing configuration files (eg. /root/AppName/config/v1.2/db.properties).

2) Key-Distribution Filter: When multiple candidate files are discovered in the above process, the TFD module uses the distribution of key names in source and target files to determine the closest target file. Standard (Unix-like) *diff*-based approaches for checking file similarity may fail here due to the difference in the value of configuration parameters between source and candidate target files. In addition, it is often not possible to rely on relative order of key-value pairs for generating a content-based *diff* due to permutation or absence of key-value tuples. Therefore, key names turn out to be the only reliable option that are mostly constant across different versions of a software.

The TFD module uses key names extracted from the source file annotations and records their distribution in a vector for each candidate file. The normalized distance between the frequency vectors for a given source file $F_S^{i,j}$ and candidate file f is combined with the respective file-path-match score to assign a total score S to each candidate file as: $S = \alpha(1 - \frac{mLen}{tLen}) + (1-\alpha)D_f$, where *tLen* is the total length of source file path, α is some pre-defined weight and D_f is the normalized distance between frequency vectors. The candidate file with the lowest combined score is selected as the match for the given source file.

D. Target PoV Discovery

The TPoD module searches for misconfigured or missing PoVs in each target file $F_T^{i,j}$ using the annotated tree of the respective source file $F_S^{i,j}$. These annotations identify the nodes representing PoVs in the source tree and other contextual information such as nodes containing keys along with the types (IP, FQDN, Number, Path etc.) of their respective values contained in sibling nodes.

1) Hierarchical File Representation: TPoD first derives an equivalent tree representation of the target file to define parent, child and sibling relationships between the constituent nodes. Besides facilitating structural comparison of the two trees, these relationships help to identify the context in which the PoV values must occur in the target file. This is necessary as the source and target files often contain the exact (or nearly identical) key names corresponding to a PoV but with different respective values. It is important to note that while the standard *diff*-based approach to patch target files can handle permutations and missing parameters, it can not deal with modifications of key names (although rare). Furthermore, all the parameters from the source file need not (or must not) be copied over to the target file to preserve application functionality, as many parameters are installation- or image-specific, such as the install path.

Given the source and target tree representations, TPoD attempts to trace the path to each PoV node in the target tree utilizing the corresponding path obtained from the source tree. Starting from the tree root, each node in the source path must be mapped to its respective node in the target path. If the complete path exists in the target tree, then the PoV must be reconfigured to its correct value. Otherwise, the last node in the target path, which matches with the corresponding node in the source path, is identified for adding the missing sub-tree containing the PoV.

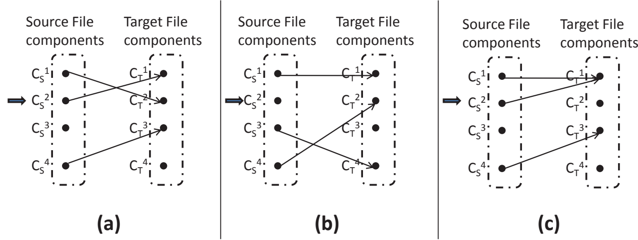

Fig. 4. XML component mapping scenarios at a given level (height) in source and target trees: (a) unique mapping (b) mapping absent (c) ambiguous mapping.

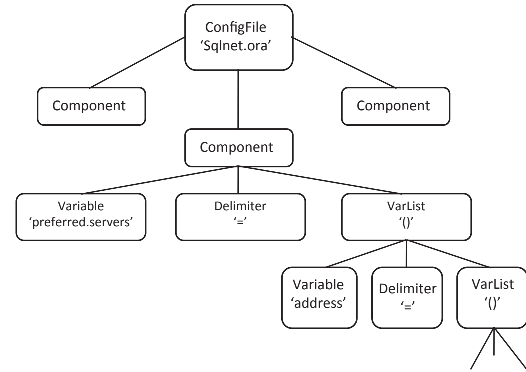

Fig. 5. A part of AST for Oracle8 configuration file shown in Figure 1.

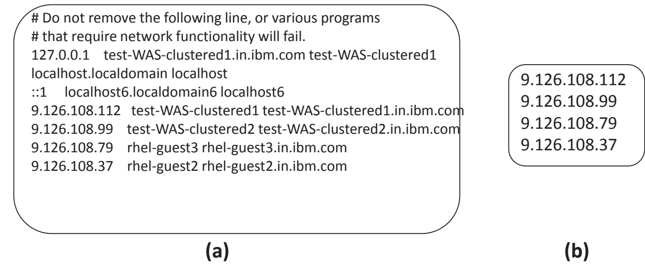

Fig. 6. Exceptional non-XML configuration files: (a) /etc/hosts (Linux) (b) Hadoop server.

2) Configuration Node Mapper: The mapping from source to target nodes is non-trivial due to the fact that nodes are neither associated with unique identifiers nor can their absolute order of appearance in the configuration files be used to assign one. Moreover, unlike solutions for graph isomorphism, the tree structure *alone* is not sufficient to derive accurate node mappings because

1) the subtrees rooted at sibling nodes often have identical structure, which results in ambiguity, and
2) in case of missing PoV components from the target file, the tree structure of the file is different, which results in incorrect decisions.

Consequently, structural information must be combined with contextual information from the configuration trees to obtain correct node mappings. TPoD employs a standard XML parser (Python *lxml*) to load an XML configuration file into its equivalent tree representation. For non-XML files, including PHP, Python and user-defined formats, TPoD uses a generic *FileParser*. The generic FileParser developed in [6] was preferred over script-specific parsers available for PHP, Python etc. to eliminate dependence over multiple third party modules (for robustness) at the risk of loss of script-specific knowledge.

XML Configuration Files: TPoD handles XML files separately from non-XML files due to the difference in their tree structures and node properties/attributes. It assigns a *fuzzy* similarity score to each source and target node pair at a given level in the trees using their tag names, attribute names and values[1]. A mapping is established between a source and target node pair when its score satisfies a certain configurable threshold. However, due to the typical structure of XML configuration files where identical tags and attribute names are used to define multiple components within a single file, we often observe deviations from the expected one-to-one mappings as shown in Figure 4. Subsequent ambiguity resolution uses limited structural information from the tree (eg., number of children) and attempts to recursively map the children of ambiguous node pairs using the same process. Our results demonstrate that the ambiguity is successfully resolved in all the cases considered in the experiments.

Non-XML Configuration Files/Scripts: TPoD decomposes a non-XML target file into an Abstract Syntax Tree (AST) consisting of *Components*, *Variables*, *Delimiters* and *VarList* nodes as shown in Figure 5. Each component consists primarily of a key-value tuple representing a PoV with the possibility of several related key-value pairs grouped into a complex value (*VarList* node, eg. preferred.servers in Figure 5). Consequently, the resulting ASTs are short but wide with many Component nodes.

At each level in the source and target trees, Components are first matched using a similarity ratio of their *primary* keys that typically appear in their subtree as the leftmost child. While the primary keys are unique for a majority of the components, some deviations are observed in configuration files with multiple sections demarcated by a unique descriptor. In the latter cases, relative distance and ordering of components with identical primary keys is used to resolve the ambiguity. A Component with a simple PoV value found in the target is replaced completely by the respective Component from the source node. However, a PoV value embedded inside a VarList node is searched and replaced similarly in a recursive fashion.

Although not the norm, some configuration files contain *only* values separated by some delimiters. Two examples are shown in Figure 6. Since the keys are absent in such cases, TPoD utilizes contextual information, derived from the types of values (IP, FQDN, Path, Number etc.), and structural information such as number and relative order of children to map nodes from source to target. This information is combined into a *fuzzy* score for each node pair and the mapping is established for node pairs with the highest scores.

IV. Experimental Evaluation

We conducted a large number of experiments to study the effectiveness of *Morpheus* in automatically configuring application instances using an annotated example. In this section, we first describe the experimental setup and methodology used for this study, followed by the evaluation metrics and results.

[1] Only non-PoV values add to this score when they are similar in both the files.

Apps	Images	Number of Configuration Files				
		Same Location		Different Location		Missing
		Same Name	Diff Name	Same Name	Diff Name	
Hadoop	Hadoop-slave1	4	-	4	-	-
	Hadoop-slave2	6	1	-	-	-
	Hadoop-slave3	2	1	4	-	-
	Hadoop-master	10	-	-	-	-
Olio	Olio-database	29	-	-	-	1
	Olio-faban	10	-	-	-	-
	Olio-app	5	-	1	-	4
RUBiS	Rubis-EJB	15	-	1	-	-
	Rubis-database	8	-	9	-	-
	Rubis-HTTP	5	-	1	-	-
Trade6	Trade6-was1	4	-	5	-	-
	Trade6-was2	4	-	5	-	12
	Trade6-db	7	-	-	-	-

TABLE II
INPUT CONFIGURATION FILE CHARACTERISTICS OF APPLICATIONS IN REAL CASE STUDIES.

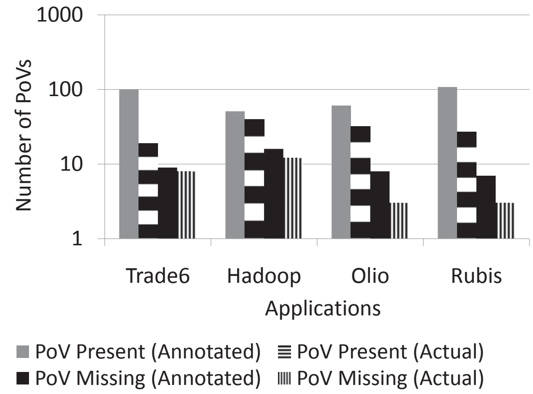

Fig. 7. Number of PoVs present and absent in the target instances of our application case studies. The annotated count is slightly more from actual count due to the presence of PoV values in some log/doc files.

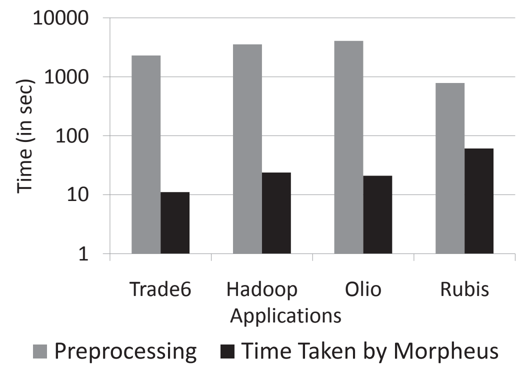

Fig. 8. The end-to-end latency of configuring each application can be decomposed into two components: (1) Preprocessing time for discovering text files and (2) actual time taken by TFD and TPoD modules in *Morpheus*.

A. Experimental Setup

Applications: Table II lists four representative benchmark applications that were selected for evaluating *Morpheus*. Trade6 and RUBiS are representative of popular multi-tier web-based applications. In the experiments, Trade6 is installed on a three node cluster, which consists of one database VM hosting IBM DB2 v8.2 and two VMs running IBM Websphere Application Server (WAS) v6.1. All VMs contain the RHEL 5.6 operating system. RUBiS is deployed as a 3-tier application with one VM hosting the $MySql$ 5.5 database, one VM running $Tomcat$ 6.0 application server and one VM running an HTTP server. Olio is used as a representative of emerging social media application deployed with a $MySql$ 5.5 database, a $Tomcat$ 6.0 application server and a $Faban$ 1.0 workload driver. The fourth application, $Hadoop$, represents a big data cloud application that is deployed with one master node and multiple slave nodes in our experiments. RUBiS, Olio and Hadoop VMs are running the $Ubuntu$ 10.04 operating system.

Experimental Methodology: Our experiments use an example installation of each application running in our lab data center. Although PoVMiner [6] is employed to annotate the sample application with only environment-related PoVs like IP addresses, ports etc., it is important to note that *Morpheus* can handle other configuration types as well. The output of PoVMiner is also fixed manually wherever required to remove any inaccuracies in the input to *Morpheus*.

Characteristics of the applications are tabulated in Table II while the identified PoVs are illustrated in Figure 7. In the first set of experiments, we try to (re)configure the target instances of each application and test their operation in a different environment. These test scenarios represent rapid application provisioning using Golden Masters and trouble-shooting of misconfigured instances. We call these set of experiments as *Real Case Studies*. For our second set of experiments, we permute the content of identified target configuration files and then randomly select file components for removal. We next run *Morpheus* on the modified files and evaluate its ability to automatically configure them. These sets of experiments are defined as the *Synthetic Worst-Case Experiments*.

B. Metrics

We evaluate the performance of *Morpheus* in terms of three metrics:

- Accuracy of File Discovery: It is measured as the percentage of number of configuration files in source application instance (A_S) correctly mapped to their equivalent files in A_T or identified as absent.
- Accuracy of PoV Identification: It is measured as the percentage of PoVs from source configuration files correctly identified as misconfigured or absent in the equivalent target files.
- Latency: It is measured as the time taken by *Morpheus* to discover target configuration files in all the images constituting an application instance A_T, identify PoVs inside them and perform appropriate action to remediate them.

C. Real Case Studies

We use three different case studies for the real-life experiments. Table II displays the variation in location, names and existence of input configuration files for the four applications in consideration. Our results confirm that *Morpheus* is resilient to these variations as the accuracy of file discovery and PoV identification is observed to be 100% in all the following cases.

1) Sanity Check Experiments: The first experiment is used to verify that all components of *Morpheus* work as expected. Our data center contains a few master images that are pre-installed with RUBiS and Olio. We select an instance of RUBiS that was installed in November 2011 and an instance of Olio that was configured in July 2012 as example instances, and annotate them with PoV values. The master images are then used to create additional instances A_T of RUBiS and

Olio. Our results confirm that (1) *Morpheus* successfully configured the new target instances and (2) the application resumed normal operation in the target environment. Specifically, *Morpheus* is able to identify 19 POVs in the target instance for Olio and successfully map the location of 3 missing POVs (Figure 7). Similarly, for RUBiS, it is able to identify 27 POVs and find the location of 3 missing POVs.

Latency: The total time taken to configure the target RUBiS and Olio instances is observed to be 68 and 59 minutes respectively. However, a majority of the time (ie. more than 99%) for processing an image I_T^i is spent in crawling through its files to separate text (ASCII) files from the binary ones (Figure 8). As a result, the actual time taken by TFD and $TPoD$ modules in *Morpheus* is 21 and 25 seconds respectively (Figure 8).

This is an artefact of the current implementation where a single thread processes each image (approx. 89,000–151,000 files) before invoking the TFD filters. It can be trivially improved by employing multiple worker processes per image (by appropriately dividing the file space) or searching through multiple images in parallel. In fact, the TFD module utilizes the former practice for filtering the discovered text files (approx. 37,000–108,000 per image) to obtain the desired result. The processing times displayed in Figure 8 for finding configuration files are obtained from experiments spawning 8 worker processes on an 8-core processor.

2) Rapid Application Provisioning: Our second case study captures rapid application provisioning. In this scenario, we instantiate new copies of the application by (i) creating VMs from golden masters that have the operating system and middleware stack pre-installed, (ii) installing the target application and (iii) configuring the target application using an annotated source instance. For the experiment, the source instance is selected as a working Trade6 application on a clustered WAS with DB2 as backend database. The target instance is created by installing Trade6 on images derived from golden masters that have pre-installed WAS and DB2. *Morpheus* is used to configure the target instance automatically. This scenario also requires the middleware to be configured. Consequently, there were a large number of missing configurations due to the unconfigured middleware. However, *Morpheus* was able to successfully configure both the WAS middleware and the Trade6 application. It identified 19 POVs in the target instance and also successfully mapped the location of 8 missing POVs (Figure 7). The end-to-end latency was 14 minutes with the actual time taken by *Morpheus* at only 1 minute. Compared to this, it took us 8 hours to configure the application and middleware for the first time.

3) Peer Configuration: Our last case study captures the scenario where multiple instances of the same application are hosted in the cloud. However, in this case, one of the instances was not running due to a mis-configuration error. In this experiment, we demonstrate that *Morpheus* is able to to remediate the failed instance and configure it using a previously annotated instance. The two instances of Hadoop used in this experiment had been created completely independent of each other. The source instance is configured with 3 slave machines while the target instance is configured with 12 slaves. The experiment attempts to automatically configure the target instance to use only 3 (of the 12) slaves. We observe that *Morpheus* is able to automatically configure the application, even with such variations in the source and target instances. It identified 40 POVs in the target instance and also successfully mapped the location of 12 missing POVs (Figure 7). The end-to-end latency was 38 minutes with the actual time taken by *Morpheus* at only 11 seconds.

Applications	Files	Total number of PoVs	PoVs identified in target files		
			Unmodified	Permuted	Deleted
Custom Configuration File					
Cassandra	1	4	4	4	4
Mongoose	1	2	2	2	2
MySQL	1	2	2	1	1
OpenVPN	1	3	3	3	3
Oracle8	5	20	20	18	18
PostgreSQL	2	5	5	5	5
TPCW	1	3	3	3	3
Transmission	1	3	3	3	3
XML Configuration File					
Jboss	1	4	4	4	4
Nlog	1	3	3	3	3
Tomcat	1	4	4	4	4
WAS	2	26	26	26	26
PHP Scripts					
Mediawiki	1	3	3	3	3
Olio	1	6	6	2	2
RUBiS	2	6	6	5	4
Wordpress	1	3	3	0	0
TOTAL		**97**	**97**	**86**	**85**

TABLE III
ACCURACY OF TPOD WHEN CONTENT OF CONFIGURATION IS (A) UNMODIFIED, (B) RANDOMLY PERMUTED AND (C) RANDOMLY DELETED

D. Synthetic Worst-Case Experiments

Our case studies established the effectiveness of *Morpheus* to automatically configure new application instances for all the motivating scenarios. However, given the limited number of real cases at our disposal, we also conducted synthetic experiments. These experiments allow us to study the limits of *Morpheus* by analyzing its sensitivity to permutation and absence of content in target configuration files. In this section, we evaluate the TPoD module in *Morpheus* on the basis of its accuracy in PoV identification. The analysis of Target File Discovery is omitted for brevity as its accuracy is observed to be 100%.

The experiments use 23 different pre-annotated configuration files from 16 web applications listed in Table III. In the first set marked as *Unmodified*, the target configuration files are simply cloned from the source input files, while in the next two sets of experiments, their contents are randomly permuted and then deleted. The resulting three sets of configuration files are processed through the TPoD module to generate the results shown in Table III. The following analysis explains these results by categorizing the configuration files on the basis of their formats.

1) XML Configuration Files: In all four applications utilizing XML configuration files, the accuracy in identifying PoVs is 100% in the three sets of target configuration files. This demonstrates that TPoD is resilient to permutation and absence of content due to the well-defined and consistent format of XML files combined with the use of an XML-specific parser.

2) Custom Configuration Files: The custom configuration files typically employ user-defined formats consisting of key-value tuples with slight variations in their exact syntax (Figures 1 and 9). Although there are some deviations to this rule, such as those shown in Figure 6, TPoD is mostly accurate in processing these files due to our carefully designed FileParser module. The results in Table III confirm that its accuracy is 100% in all three sets of configuration files for 6 out of 8 applications that utilize custom formats. However, for the

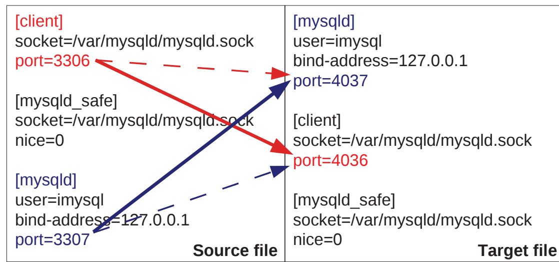

Fig. 9. Snippets of MySQL configuration file from source and target application instances. The contents of target file are rearranged. The correct mapping of *port* PoVs is shown with solid arrows. The section markers in the files are surrounded with "[]".

remaining two applications, the accuracy drops for *Permuted* and *Deleted* sets with the exact deprecation depending on the content affected by the random permutation and deletion operations. Table III presents results for the cases which incurred the lowest accuracy.

The primary reason behind these errors is the dilemma of assigning appropriate importance to the structural information encoded in the configuration files. While we claim that, in general, permutation and absence of content render the file structure as unimportant, we observe that the structure can occasionally provide crucial information for resolving an ambiguity that can not be resolved by context alone. For instance, in Figure 9, the *port* PoVs in source file can only be mapped to the correct PoVs in the permuted target file if the relative position of the PoV nodes with respect to the section markers (and/or other sibling nodes) is utilized to break the tie. This dilemma also fuels our hesitation in using global structural information, similar to graph isomorphism, rather than the current approach of utilizing localized child index and relative ordering from the tree. We are actively working on resolving this issue in the near future.

3) PHP Configuration Scripts: The TPoD module displays relatively lower accuracy in identifying PoVs in *Permuted* and *Deleted* sets of target PHP configuration scripts. In addition to the global structural information that is not used by TPoD, PHP scripts can not be processed accurately due to the loss of language-specific information through the generic FileParser module as well. For instance, the definition of multiple key-value pairs using *dictionaries* such as "*$olioConfig[key]=value*" results in incorrect node mappings as the correct "key" (context) can not be associated with the respective value (instead, the commonly occurring "$olioConfig" is associated with the value). However, we expect these errors to be fixed automatically after we incorporate additional structural information in TPoD as mentioned above.

V. Related Work

The related work in area of automatic application configuration can broadly be classified into

Configuration Discovery Configuration discovery involves inferring the dependencies between multiple components of an application. The Galapagos [12] system uses network profiling to automatically discover relationships between application and data. PoVMiner uses data mining techniques to discover the configuration information in applications in an oblivious manner [6]. All these techniques can be used to annotate configuration information in an example instance, which can be leveraged by *Morpheus* to configure other instances in the same or different environment.

Automated Reconfiguration IT Service Automation has been a goal with the aim of reducing IT service delivery cost and improving reliability. Researchers have looked at IT Service automation frameworks [5], which combine manual and automated steps. Automated steps may include software install, whereas configuration steps are typically manual or based on catalog for known software. Work in automating individual steps based on a catalog has been a popular technique in the industry. Mastrianni *et al* migrate applications and data from one environment to another, where metadata for each application is used to identify the files belonging to an application [13]. [10] creates a script based application that contains the logic of changing the configurations in the target environment. In [11], the authors propose a model-based dependency management framework that uses scripts for configuration automation. Similar efforts have created deployment models for a multi-component application [9]. Capps *et al.* migrate system settings and configuration information (e.g., display settings, power policies) from one personal computer to another [8]. All of these techniques leverage expert information about the underlying application or the operating system making them unsuitable for reconfiguring custom applications. *Morpheus* fills this gap in automated application configuration by allowing custom applications to be configured based on an annotated example.

Auto-reconfigurable Software Development: Distributed applications that are easily reconfigurable has been a lofty goal in the domain of distributed systems. *Plush* allows developers to specifically define the flow of control needed by their computations using application building blocks [4]. Olan provides a model for developers to specify components and communication models [7]. The advent of virtualization has enlarged the scope of configurable software from middleware to the entire software stack including the operating system. This concept has been termed as Virtual Appliances [15], which is a network of virtual machines pre-installed and pre-configured with complex stacks of software. Virtual appliances also are restrictive requiring them to specify configuration and communication in standardized way. It is possible that the restrictions posed by configurable middleware and appliances have prevented widespread adoption of appliances.

VI. Discussion and Conclusion

In this paper, we presented a tool *Morpheus* for automatically reconfiguring the applications to work in a new environment. *Morpheus* uses a configured instance of a workload, annotated with the configuration information, to automatically configure new instances of the workload. We used *Morpheus* to deploy new workloads for four multi-tier enterprise/web applications, namely Hadoop, RUBiS, Olio and Trade6. It was able to deploy them without any manual intervention in 13–68 minutes for 100% of the workloads.

However, it occasionally failed in synthetic worst-case experiments when ad-hoc structures were present in unstructured files with randomly permuted and missing content. We aim to eliminate these failures in the near future by better utilizing the configuration file structure to resolve the ambiguities in mapping PoV nodes. Another limitation of the system is that it cant interpret configuration hard-coded in binary files.

REFERENCES

[1] Amazon Elastic Computer Cloud (EC2). http://aws.amazon.com/ec2.
[2] Crimson Consulting Group White Paper: Cost of Ownership Analysis. http://www.oracle.com/us/products/middleware/crimson-weblogic-websphere-tco-402458.pdf, 2011.
[3] Crimson Consulting Group White Paper: Cost of Ownership Analysis. http://www.oracle.com/us/products/middleware/application-server/weblogic-vs-jboss-460235.pdf, 2011.
[4] Jeannie Albrecht, Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Remote control: distributed application configuration, management, and visualization with plush. In *Proc. of LISA*, 2007.
[5] Naga Ayachitula, Melissa J. Buco, Yixin, Maheswaran Surendra, Raju Pavuluri, Larisa Shwartz, and Christopher Ward. It service management automation - a hybrid methodology to integrate and orchestrate collaborative human centric and automation centric workflows.
[6] R. Balani, D. Jeswani, D. Banerjee, A. Verma, and K. Bhattacharya. Povminer: Application-agnostic configuration mining. In *IBM Technical Report*, 2013.
[7] R. Balter, L. Bellisard, F. Boyer, M. Riveill, and J.Y. Vion-Dury. Architecturing and configuring distributed application with Olan. In *Proc. of Middleware*, 1998.
[8] S. P. Capps, D. A. Feinleib, J. D. Swed, and D.S.Johnson. System and method for the automated migration of configuration information. In *US Patent 6,735,691*, 2004.
[9] T. Eilam, M.H. Kalantar, A.V. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal. Managing the configuration complexity of distributed applications in Internet data centers. In *IEEE Communications Magazine*, 2006.
[10] A. Ganguly, J. Yin, H. Shaikh, David M. Chess, T. Eilem, Renato J. O. Figueiredo, J. Hansom, A. Mohindra, and Giovanni Pacifici. Reducing Complexity of Software Deployment with Delta Configuration. In *Proc. of Integrated Network Management*, 2007.
[11] Qian Ma, Ying Li, Kewei Sun, and Liang Liu. Model-Based Dependency Management for Migrating Service Hosting Environment. In *Proc. IEEE SCC*, 2007.
[12] Kostas Magoutis, Murthy Devarakonda, and Kiran Muniswamy-Reddy. Galapagos: Automatically discovering application-data relationships in networked systems. In *Proc. of IEEE IM*, 2007.
[13] S.J. Mastrianni and T.E. Chefalas. Method and apparatus for the automatic migration of applications and their associated data and configuration files. In *US Patent 7,028,079*, 2006.
[14] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. In *NIST Special Publication 800-145*, 2011.
[15] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum. Virtual appliances for deploying and maintaining software. In *Proc. Usenix LISA*, 2003.
[16] Deepak K Vij and David Bernstein. Cloud to cloud interoperability and federation - intercloud. In *IEEE P2302l/D0.2 Draft Standard for Intercloud Interoperability and Federation (SIIF)*, 2012.
[17] VMWare, Inc. Managing Multi-Tiered Applications with VMware vApp. vSphere Virtual Machine Administration Guide http://pubs.vmware.com/vsphere-4-esx-vcenter/index.jsp?topic=/com.vmware.vsphere.vmadmin.doc_41/vsp_vm_guide/managing_vms_with_vapps/c_managing_vmware_vapp.html, 2011.