Technical Report

TR-2014-006

Multi-Target Deployment for eScience Applications

by

Jaroslaw Slawinski, Vaidy Sunderam

Mathematics and Computer Science

EMORY UNIVERSITY

Multi-Target Deployment for eScience Applications

Jaroslaw Slawinski, Vaidy Sunderam
Mathematics & Computer Science Department, Emory University
Atlanta, USA; {jslawin, vss}@emory.edu

***Abstract*—Streamlined switching between computational resources in order to select the most suitable computational environment for application execution is a crucial component of cloud-like computing. However, heterogeneity obstructs multi-target deployment for complex and multi-dependency scientific and engineering codes and makes this goal intractable. In this paper, we describe a proposal for a metadeployment toolkit, called ADAPT, based on reusable recipes that address appropriate match-up between an application and an execution platform. Our research aims at exploring challenges posed by automatic deployment of applications with all their prerequisites on heterogeneous resources. As some IaaS clouds and grids accept customized OS images, we explore application-oriented image assembly to improve deployment for these targets. We explain how our approach increases "usability" of various resources and simplifies arcane build of eScience applications.**

***Index Terms*—heterogeneous targets, deployment automation**

I. Introduction

The recent surge in on-demand computational offerings encourages scientists to experiment with eScience apps on a broader assortment of resources. Despite unavoidable performance degradation in some cases, new platforms attract for various reasons: instant resource availability (no job queues; any number of hosts), cost (no upfront expenses), or greater control (root privileges or direct access). However, users cannot rely on the user support, which is customary at HPC centers—traditional execution environments for eScience. Occasionally, eScience software requires porting to enable execution on a particular resource or, reversely, the available code of apps may be tuned to hardware-specific capabilities absent on common platforms. Clearly, more *autonomous* approach in app deployment and an improved execution model are needed; however, these subjects in the context of eScience apps have not attracted sufficient research attention yet and solving dependency related issues usually imposes an unnecessary burden on scientists who may be domain experts in, e.g., physics or biology but not necessarily in computer science.

This paper delivers a proposal of a deployment system that provides an automatic, adaptive, and transparent multi-target deployment solution. The design employs reusable *deployment recipes* that capture expert knowledge related to software conditioning. Proper chaining of these recipes formulates *automatic deployment scripts* that probe and soft-condition a target environment *until* the considered app is successfully installed. As a result, users may execute their apps on a wider range of resources without drudgery pertained to deployment phases. Our proposal not only enhances usability for computational offerings but also has the potential to increase productivity in HPC, by providing systematic and automatic environment conditioning. Moreover, this approach supports know-how sharing beyond a narrow group of specialists. We address a broad spectrum of machine architectures, from computational clusters, which are optimized by-design for eScience, to single hosts, Grids, and IaaS clouds, to PaaS services. We believe, that smooth switching between computational targets, even for a single app run, may lay a foundation for the long-standing goal of Computing-as-a-Utility.

Research supported in part by NSF Grant OCI-1124418

II. Related Concepts

The excessive effort related to software deployment is a recurring motif in many projects [27]. Typical *build automation software* (e.g., GNU Make [7], CMake [3], Ant [1], SCons [13], Gradle [8]) is designed to build a single, yet possibly multiproject, software bundle, such as an individual library or executable. Such tools target specific programming languages, such as C/C++, Fortran, or Java, and may require dependencies that are uncommon in typical eScience environments. Moreover, locking onto a particular build automation tool may require porting of the existing build solution for some dependencies of a particular app. We propose a more general approach based on the concept of *metabuild* that enables dependencies in their native deployment formats. Our metabuild aims at *management* of existing deployment methods.

The installation of software packages with automatic dependency resolution is addressed by *package management systems*, such as RPM [12] or various implementations of Ports [9], [10]. Selected software is delivered in a form of standardized packages downloadable from repositories. The package includes the software payload (either precompiled or as source code) and a dependency inventory used by an automatic dependency resolution mechanism. However, even if eScience apps are distributed as standardized packages, their up-to-date versions are rarely supported by maintainers and, in practice, the software has to be built from the source code. In addition, the build is often proprietary, which may greatly hamper unification in a chosen build format: the software often requests specific versions and *selective compilation* or patching of its dependencies [23]. The way the package managers automate installation prevents simple maintenance of many versions of the same libraries (file conflicts) and, in practice, disables selective tuning. These issues deter use of common package systems as an exclusive solution and any eScience app deployment toolkit must address extraordinary

app build requirements. However, we intend to use target-specific package systems if they are available, as they simplify dependency provisioning.

Executing an app on IaaS clouds may seem to be a simple task as the user can entirely shape execution environments of instantiated virtual hosts. However, only a fraction of cloud providers allow users to run an arbitrary OS, whereas remaining IaaS platforms impose a limited selection of OS's. This establishes secondary issues related to heterogeneity in the system software and, as the result, such clouds do not differ much from other targets. On the other hand, customized images significantly improve software provisioning as a client may bundle the entire software stack within the image. Projects such as OSCAR [21] or rPath [18] deliver customized OS images with requested software preinstalled from package repositories. However, as these images are based on standard distributions, they easily become overinflated in terms of required storage and provided services, which may lead to increased upload overhead and unnecessary service cost. In order to mitigate this problem and improve the computational efficiency, we propose another solution for assembling images for IaaS. Our metabuild generates *images for a single run* of an app using a minimalistic Linux distribution for a specific IaaS platform–app pair.

EasyBuild [20] targets installation of eScience apps by standardization of common deployment steps, such as downloading, configuring, or compilation. Each software component handled by EasyBuild is represented as an extensible Python component implementing all steps required to activate the software. Further declarative specialization of this script delivers deployment details for specific software versions and configurations, such as a required compiler toolchain. EasyBuild extensively uses Environment Modules [6] to "register" installations and resolve dependencies. Our approach, instead of templatizing and standardizing of deployment steps, promotes capturing native commands that users routinely perform to install any software. Such mapping is more natural for the users as they may immediately preserve their pragmatic, software installation-related knowledge. We do not impose any formalism; users are free to provide their deployment steps in the form of a snippet in their favorite script language and *tag* it for the future reference and use in other metadeployment scripts.

Configuration management (CM) tools, such as Puppet [22], Chef [2], or Sprinkle [14], are often used to provide a well-defined set of software on multifarious targets and support variety of OS's, including the Windows family, in a transparent way. Such tools are beneficial for the system admins who can automatically and in a reproducible way deploy software on hosts inside their administrative domains. For these reasons, CM tools are frequently recommended to the users by IaaS cloud providers who are unwilling to offer support for customized images. The primary advantage of such projects is that the user may define the software stack in a *semi-declarative* way using predefined actions and properties. CM's may provide an additional features, even if it breaks the overall portability, as bare command execution and mix command-based operations with other built-in deployment steps. Variety of eScience software deployment issues causes that the command execution capability must be used for eScience software, which weakens usability of CM's as homogeneous solutions for many targets. Our tool elevates *command execution* and *standard output streams* processing to key components of the system: the metadeployment scripts are chains of independent, task-oriented recipes driven by the error status of previously executed commands.

III. ADAPT Overview

The primary goal of this project is to extend the usability of hardware architectures by enabling execution of *unmodified* eScience apps with the assistance of the *adaptive middleware* ADAPT (ADaptive Application and Platform Translation). Apps may be executed on different targets thanks to the ADAPT middleware that generates *situation specific*, app–target adapters [17].

ADAPT proposes a simple model of app execution: in order to sustain an app (support its run-time), all *application requirements* have to meet their corresponding *resource capabilities* on the selected target. The requirements are recognized as software dependencies (e.g., compatible routines stored in dynamic libraries), binary compatibility, communication or interaction capabilities, etc. The resource capabilities are all capabilities offered by the resource and interfaced by its system software such as storage, e.g., local file system, inter-process communication, e.g., present network fabric, or computation, e.g., opcode sets, concurrency support. The ADAPT middleware performs bidirectional coupling by applying *software environment conditionings* to enhance resource capabilities as well as modifying apps requirements to match to actual resource capabilities—the same app may require a different set of adapters in order to be executed on a different resource.

Note, that the app requirements remain constant whereas the target capabilities vary from machine to machine. As a result, one target may be ready to build the app instantly, while another target must undergo multi-stage software conditioning in order to meet the requirements. In order to provide flexibility, the most suitable method to create an adapter is to extend the resource capabilities by applying additional software layers; in extreme cases, the missing resource capabilities may be virtualized, e.g., by creating a virtual machine, providing a dynamic binary translation or emulation or outsourced, e.g., providing permanent storage capability on a diskless host. By extending capabilities rather than modifying requirements, our approach performs the bottom-up adaptation.

In this paper, we focus on provisioning the eScience software on various targets. The issues related to data staging in/out, launching, and monitoring are beyond the scope of this consideration. We believe that deployment should be fully automatic; thanks to that, the user experiences no operational difference between various targets which greatly homogenizes use of different offerings. Further, this promotes experimentations with eScience apps on targets differing from well-

supported, typical execution platforms. To implement this vision, ADAPT applies software components on resources in a layer-by-layer fashion until the requested level of specialization is achieved.

IV. Metabuild Description

In this paper, we propose a design of a toolkit to enable deployment of apps from the eScience class onto a wider range of computational resources. The eScience soft-conditioning is particularly difficult as that apps are usually distributed in the form of source codes, require multifarious, nontrivial, and numerous dependencies as well as utilize parallel and distributed programming paradigms. Moreover, as they solve cutting-edge problems, they often require performance tuning to efficiently utilize the underlying hardware infrastructure. Our proposition enhances usability of these apps beyond on-premises or supercomputer center machines and aims at offering the software on any (parallel) architectures accessible for the user, including department clusters, grids, or IaaS clouds. We aim to embrace the heterogeneity resulting from using a variety of targets by building eScience apps from sources. As a result, our solution may extricate users from the burden related to an unproductive software deployment phase and promote switching between targets and vendors for availability or financial reasons, even for a single run of an app. Moreover, this may help popularize eScience apps beyond a close community as well as may increase the overall deployment productivity in traditional eScience environments.

a) Metadeployment Scripts: The ADAPT idea is to use generic *metadeployment scripts* that adapt their execution to a particular *deployment scenario* defined as the specific app–target pair; this concept is presented in Figure 1. A metadeployment script calls matching *deployment recipes* retrieved from a *recipe repository* and defines the correct order of dependency provisionings. The script has no explicit software dependencies—it depends only on a shell and may bootstrap its dependencies if needed. The users execute such a generic script for a given target in order to deploy their software: the script examines requirements and supplies them applying recipes. In a case of errors during recipe application, the script tries to *fix* the issue and reexecute the last command or *rollbacks* the invalid recipe and attempts to run an *alternative recipe*. On success, the deployment steps may be saved and reused for another deployment on similar targets.

As we mentioned in Section II, typically, eScience app distribution packages already include build systems based on popular toolkits, such as GNU make, CMake, or shell scripts. The build systems usually specify—more or less explicitly—requirements, that is, prerequisites such as software dependencies or required environment variables. In the context of ADAPT, the build systems enhance the target with software capabilities such as libraries, headers, or executables by building and installing particular software. Another vital but often hidden information is *metadata* related to actual system tools used to build the software, versions of such the tools, compilation flags, etc. As complex scientific software typically requires several dependencies, which may have their requirements, all deployment steps must be carefully chained and the compatible system tools must be used throughout the entire deployment process.

We emphasize that since apps already come with own deployment methods, we do not aim at providing yet another build system; instead, in order to automate the deployment we propose to: (1) abstract different deployment methods, (2) provide straightforward *chaining* of currently separated and often manually executed steps, (3) keep track of metadata to enforce compatibility between software components, and (4) monitor deployment to detect errors and fix them. This will help make deployment knowledge *reusable*, *sharable*, and *self-documenting*, and increase productivity in eScience.

b) Target Environments: eScience apps are typically designed for classical computational resources, viz. workstations, clusters, and grids. For those systems, an access privilege level shapes possible interactions between users and a deployment environment. As a rule, users have limited access to computers and perform software conditioning in the *user space*, with consequences resulting from those limitations, or have to ask the site admins for support. As our method needs to be transparent and universal, we shun any external support and focus on provisioning in the user space; however, this does not exclude work in the elevated privilege modes if possible.

The infrastructure clouds can be easily specialized with the use of *successive conditioning* [26] that yields chunks of classical resources on-demand so software provisioning may be easily performed on them in a traditional manner. However, as several IaaS cloud and grid providers allow the users providing their operating system images, what gives superior flexibility even in comparison to privileged access, we put forward a more specialized approach. Instead of applying software conditioning on virtual resources running a standard OS, we will *generate* an OS image that (1) is tailored both for the app and the underlying virtualized platform and (2) can be formed for single execution of a given app. As such the design may ignore objectives other than the app execution, i.e., we can reduce the system software to bare, essential functionalities that are necessary yet sufficient to sustain the execution. This significantly reduces both the *size* of the image and the *operating system noise* [19]. Thanks to this reduction, we intend to improve execution performance, decrease the image upload and boot time, thus lowering the resource utilization costs. Thus, execution of a program would be as simple as sending an image file to the target and staging out the results.

c) Operational Scenarios: To deploy an app, users run its metadeployment script on a selected target and the script conditions the execution environment using an appropriate set of recipes, as it is depicted in Figure 2. In the more general scenario, users may specify a target different from the current one which initiates “cross-deployment”; thanks to that, ADAPT may address soft-provisioning of compute nodes in computer clusters that do not share the situation-specific OS images for resources that accept custom images. As the latter topic is more interesting from our research

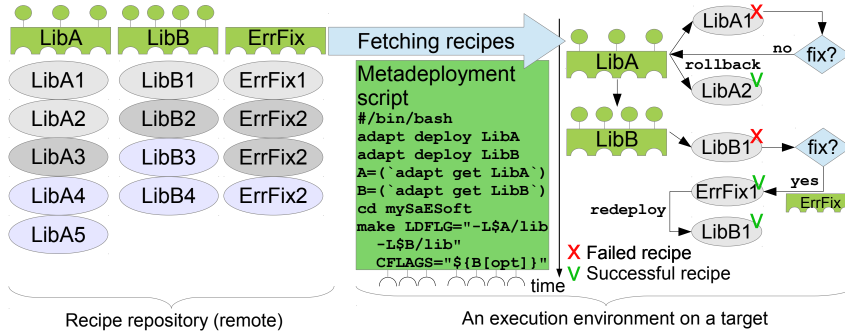

Fig. 1. A generic metadeployment script fetches and applies recipes. It may recover if deployment errors happen. The power of ADAPT is alternative recipes.

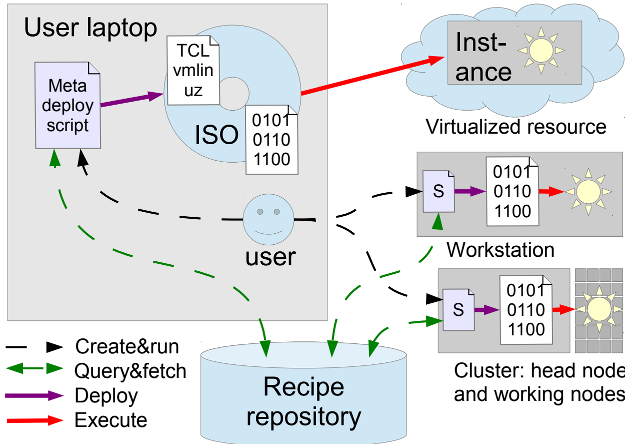

Fig. 2. Users create and execute metadeployment scripts that install software. App-oriented images are generated for targets accepting customized images.

standpoint, we plan to study this option more carefully. So far, we have experimented with Tiny Core Linux [4] that can be remastered to meet specifics of a virtualized platform and tuned to support performance (e.g., swapping disabled). To avoid superfluous library payload, we intend to build an app *statically*. Consequently, this may limit the binary requirements to syscalls, thus benefiting performance. Finally, proper configuration of the image may cause that the app will start when the instance boots (`init`) and the instance may be terminated on the app's conclusion. In addition, the user *can* request extra services, such as `sshd`, to supplement the execution with monitoring or control.

V. Metadeployment Design

The goal of ADAPT deployment is to keep metadeployment scripts *generic* so a single deployment script may serve to a broad assortment of different platforms and their heterogeneous configurations. Our solution for that is to (1) offer *target-agnostic* deployment commands for metascripts and (2) provide *alternative* implementations of particular software deployments (recipes) that may differ for specific targets and situation-specific configurations. As a result of such the general for users and specific for targets approach, we need a proper (3) identification, classification, and searching mechanisms to couple the metascript deployment calls with valid recipes. This is equally important to make this solution automatic and autonomous to promote the computational resource usability—for this reason, we also propose (4) an automatic solver for problems that might arise during deployment processes. The later element allows also relaxing the design requirements: instead of providing highly detailed target- and recipe-related descriptions for error-avoiding recipe coupling mechanisms we propose an error-aware approach that monitors deployment steps and fixes issues as they occur. The following paragraphs describe our approach to address the aforementioned issues in greater detail.

d) Metadeployment Script: eScience software deployment is complex and requires trained specialists to (1) recognize and provide software prerequisites and (2) conduct the actual build steps followed by the software component activation. However, all human–build environment interactions may be *captured* and stored as scripts. Based on these observations, we propose a twofold design. The first, target-agnostic *header* secures all direct software requirements, i.e., determines locations of already installed prerequisites or deploys, using ADAPT, new software components; subdependencies are deployed recursively by ADAPT. The second, *terminal* part is app-oriented with the actual installation steps performed; here, the provided or recommended build technique is used (e.g., `./configure; make`). To parameterize this build, creators of metascripts can *query* ADAPT for capabilities of deployed software dependencies such as locations of installed libraries or toolkits used to compile the dependency. An example of a metadeployment script is shown in Listing 1.

Listing 1. A metadeployment script for LifeV simulations [25]

```
#!/bin/bash
#HEADER: locate or install lifev
adapt deploy lifev
#TERMINAL PART: use actual build steps
# 1. query direct and indirect capabilities
lib=$(adapt get lifev.lib)
blas=$(adapt get lifev.blas.lib) ...
# access also metainfo of capabilities
export CC=$(adapt get lifev.meta.cc.path)
# 2. download requested software
curl lifev.com/aorta.tar.gz | tar xz; cd aorta
# 3. native build steps: A. create Makefile.in
sed "s/^\(LIFELIBPATH\).*/\1=`$lib/;
 s/^\(BLASLIBPATH\).*/\1=$blas/; ..."
 < Makefile.in.sample > Makefile.in
```

```
# B. build; fix errors if they happen
adapt monitor make
```

e) Recipe Design: We design the ADAPT recipes to implement abstract dependencies provisioning for metadeployment scripts. As such a task is internally recursive, we use the idea of generic metadeployment scripts for softconditioning of subdependencies. From a more abstract point of view, the recipes support the software component life-cycle; they (1) specify requirements, offered capabilities, and other deployment conditions such as target compatibility, (2) probe execution environments for the software to avoid excessive installation, (3) stage in the software package, (4) activate the software, e.g., by using the native build or loading appropriate environment modules, (5) verify correctness of the deployed software, and (6) rollback the deployed resources. As a single software component may be deployed in several ways and these methods may vary from platform to platform, each app deployment scenario may be represented by many recipes that implement these specific situations; similarly, different versions of the same software should be represented by different recipes, e.g., the download URI and deployment package change. On the contrary, ADAPT deployment procedures used within metadeployment scripts must abstract those implementation details and remain generic. To conciliate those opposing objectives, we propose the object-oriented (OO) recipe design as this is outlined in Figure 3. We introduce `DInterfaces` for abstract descriptions of capabilities and requirements of software components. A metadeployment script may query for those abstract definitions when it needs to provide required software and ADAPT methods select the most suitable recipes to execute for a given target. To promote semantic expressiveness and reusing of recipes, we permit typical OO techniques, such as inheritance or composition, to express relations among both interfaces and recipes. Thanks to that, adding a new version of an app requires a differential update; also one recipe may use the content of other recipes.

f) Recipe Repository: This is designed to store, search for, and maintain the recipe scripts. We envision that each recipe has attached metadata that describes and classifies it. We intend to use *tags* and triple tags as the mechanism to classify the recipes. Tags may characterize software provided by a particular recipe, identifying its version, dependencies, or compatible targets, such as `Atlas`, `MPI`, or `version=11.0`. Another group of tags associated with the recipes may describe compatibility of their script with targets, such as `CentOS`, `version>6.0`, `x86-64`, `ksh`, or `Nvidia-Tesla`. Also, more structured classification mechanisms such as RDF taxonomies [11] may be provided. When the `deploy` command (cf. Listing 1) fetches recipes, it may query the repository as in the following examples: (1) `adapt deploy blas goto` for the tag-based search, (2) `adapt deploy sparql:"select ?id where{ ?id a LAPACK; compatible ?soft}"` [16] for the RDF-based search. The target related tags are provided implicitly by the `adapt` command; however, the user can specify another set of target tags to control crossdeployment.

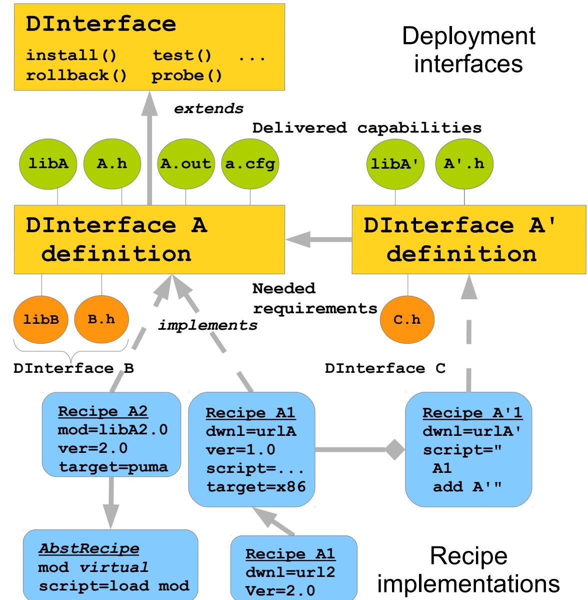

Fig. 3. The OO recipe design. The recipes are grouped by abstract interfaces that define capabilities and requirements of software components.

g) Related Issues: A single deployment step may be multistage and may need several deployment attempts before the required software is installed. To isolate different software conditioning tries, it may be required to provide a *rollback* mechanism for the resources partially staged by a failed recipe. It may require journaling of disc operations, similar to what we studied in [24], to help revert to the state existing before last changes. Also, we can use `chroot`/*sandboxing* techniques to jail the command execution for an attempt of a deployment step and "commit" the software installation on success. Sandboxing should be also considered for security reasons—as the recipes are intended to created by a community they may contain malicious snippets or unintentional errors compromising the system stability. Additional verification mechanisms for the recipe content may be considered.

VI. Tests

The first apps that we would like to equip with the ADAPT deployment are LifeV-based hemodynamic simulations (blood flow simulations). LifeV is a Computational Fluid Dynamics (CFD) Partial Differential Equation-based (PDE) library developed by our collaborators at Emory University; the details about the LifeV software and their scientific library dependencies are given in [25]. We believe that our approach will deliver an easy deployment method in a form of portable scripts for a variety of targets and enable dissemination of our simulation software beyond our local scientific community.

After prototyping, the research will continue on comparing different deployment approaches for our hemodynamic simulations using different frameworks, including Chef, Sprinkle, DoIt [5], and EasyBuild. We are interested in answering the following research questions: how the eScience app's atypical requirements may be expressed in different frameworks,

how flexible is the solution as well as what are user effort and reusability/portability of once written solutions. After collecting the experience, we plan to implement the toolkit and deliver the multi-target deployment mechanism for our in-house hemodynamic simulations. In our vision, the users interested in running their simulations with assistance of our software write or download from our repository ADAPT deployment metabuild scripts for their targets. Consequently, this requires to provide several ADAPT recipes for nontrivial LifeV dependencies. Our aim is to provide an equally easy build no matter what target was selected by the user: a local machine, on-premises cluster, or IaaS cloud. Also, we will challenge our approach with VisIt[15]. This test is interesting as the monolithic VisIt build script—a shell-based build—provides an autonomic deployment mechanism. We believe that our approach will provide an equivalent deployment with just several generic ADAPT commands and relevant recipes.

The experimental development of ADAPT is available at `http://code.google.com/p/dadapt`.

VII. Summary

Our proposal aims at providing ways for automatic software conditioning with the use of generic, app requirement-aware metadeployment scripts. App deployment remains challenging, especially if software needs to be constructed from source codes. This becomes often an extremely difficult task in the context of eScience that uses hybrid programming models, exploits various parallel processing mechanisms, and depends on multitude and precisely specified dependencies. Multiplying these issues by the number of heterogeneous targets makes this problem intractable. This is not rare when users are locked to a particular target because deployment of an app is so challenging that they cannot switch other machines; consequently, they are vulnerable to disadvantages such as low resource availability or excessive levies. These issues significantly hinders experiments with other offerings and emerging technologies as well as obstructs progress in science.

To overcome this, we sketch a design of a pragmatic, multi-target deployment system that *integrates* currently separate deployment phases for an app and its dependencies. We aim to *capture* diverse users' activities leading to an installation of a software component on a given platform and, next, to *process* and *reuse* such deployment knowledge in other deployment contexts. With respect to the computational targets, we intend to deliver a deployment toolkit for a spectrum of machines, from typical eScience machines to a single workstation and virtualized platforms. For the latter class of targets, instead of deploying the software on running instances, we intend to generate an OS image tuned for a specific scientific app–virtualized target pair. This eliminates extra deployment steps required after obtaining a generic instance from the vendors' multitenant machines or removes necessity of having a set of preconditioned OS images prepared for a particular app–machine pair. Situation-specific images also decrease a number of logical steps required to start an app to just (1) upload the image, then (2) download the results.

The grand research outcome is to support the Computing-as-a-Utility—the vision where users may select any computational resource in order to execute their apps. The Cloud Computing is the "hardware" part of this idea. The "software" part may be realized by providing tools that automatically and transparently mediates between apps and targets, even for a single run of an app. From a more abstract point of view, such automatic and autonomous deployment causes that the users are unable to functionally distinguish different execution environments and may treat other targets as a natural extension of their local computer. Consequently, this may fund a viable model of using Cloud Computing as extendable co-execution environments.

References

[1] The Apache Ant Project. http://ant.apache.org, 2013.
[2] The Chef Project. http://www.opscode.com/chef, 2013.
[3] The CMake Project. http://www.cmake.org, 2013.
[4] The Core Project. http://tinycorelinux.net, 2013.
[5] The DoIt Automation Tool Project. http://pydoit.org/, 2013.
[6] The Environment Modules Project. http://modules.sourceforge.net, 2013.
[7] The GNU Make Project. http://www.gnu.org/software/make, 2013.
[8] The Gradle Project. http://www.gradle.org, 2013.
[9] The OpenBSD Port Project. http://openbsd.org/porting.html, 2013.
[10] The pkgsrc Project. http://www.pkgsrc.org, 2013.
[11] The Resource Description Framework Project. http://www.w3.org/RDF, 2013.
[12] The RPM Package Manager Project. http://rpm.org, 2013.
[13] The SCons Project. http://www.scons.org, 2013.
[14] The Sprinkle Project. https://github.com/sprinkle-tool/sprinkle, 2013.
[15] The VisIt Project. https://wci.llnl.gov/codes/visit, 2013.
[16] W3C Recommendation: SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query, 2013.
[17] J. Bourgeois, V. Sunderam, J. Slawinski, et al. Extending executability of applications on varied target platforms. In *High Performance Computing and Communications (HPCC), 13th Int. Conference on*. IEEE, 2011.
[18] J. Craig. Cloud Coalition: rPath, newScale, and Eucalyptus Systems Partner on Self-Service Public and Private Cloud. *Enterprise Management Associates*, 2010.
[19] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing Application Sensitivity to OS Interference Using Kernel-Level Noise Injection. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 19:1–19:12, Piscataway, NJ, USA, 2008. IEEE Press.
[20] K. Hoste, J. Timmerman, A. Georges, and S. D. Weirdt. Easybuild: Building software with ease. *High Performance Computing, Networking Storage and Analysis, SC Companion:*, pages 572–582, 2012.
[21] C. B. Leangsuksun, L. Shen, T. Liu, and S. L. Scott. Achieving High Availability and Performance Computing with an HA-OSCAR Cluster. *Future Generation Computer Systems*, 21(4), 2005.
[22] J. Loope. *Managing Infrastructure with Puppet*. O'Reilly Media, Inc., 2011.
[23] M. Slawinska, J. Slawinski, and V. Sunderam. Portable Builds of HPC Applications on Diverse Target Platforms. In *Parallel & Distributed Processing (IPDPS). International Symposium on*. IEEE, 2009.
[24] M. Slawinska, J. Slawinski, and V. Sunderam. A Practical, SCVM-based Approach to Enhance Portability and Adaptability of HPC Application Build Systems. In *Int. MultiConf. of Eng. and Comput. Scientists*, 2012.
[25] J. Slawinski, T. Passerini, U. Villa, A. Veneziani, and V. Sunderam. Experiences with Target-Platform Heterogeneity in Clouds, Grids, and On-Premises Resources. In *26th Int. Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. IEEE, 2012.
[26] J. Slawinski, M. Slawinska, and V. Sunderam. The Unibus Approach to Provisioning Software Applications on Diverse Computing Resources. In *2009 Int. Conf. On High Performance Comp., 3rd Int. Workshop on Utility and Grid Comp.*, 2009.
[27] G. Wilson, D. Aruliah, C. Brown, N. Hong, M. Davis, R. Guy, et al. Best Practices for Scientific Computing. *arXiv:1210.0530*, 2012.