Approaching Utility Computing via Adaptive Multi-Target Deployment

by

Jaroslaw Slawinski, Vaidy Sunderam
Approaching Utility Computing via Adaptive Multi-Target Deployment

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

Abstract—Streamlined switching between computational resources in order to select the most suitable computational environment for an application execution is a crucial component of utility-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 transparent application deployment with all its prerequisites on heterogeneous resources. As some IaaS clouds and grids accept customized OS images, we explore application-oriented image assembly to further improve deployment for these specific targets. We explain how our approach increases “usability” of various resources for application execution and simplifies arcane build processes.

Index Terms—utility computing, cloud computing, computing grids, HPC application deployment, scientific applications

I. INTRODUCTION

Our vision of Utility Computing is to some extent contrary to the common understanding of this term [23]. Instead of improving interaction models by making them either more general or more specialized [1], [2] or tweaking existing services to better support user needs, we focus on pure computational usability of computers, i.e., if a given target can sustain execution of an unmodified application. For such arbitrary application–machine coupling, only few platforms may instantly host an application; most such pairs require software stack setup (software conditioning) and/or modifications of the application (porting). To make this model utility-like, we postulate providing software middleware that automates application–target matching and enables straightforward execution of a particular application on a chosen platform. Thanks to such transparent support, switching between computational power providers may be as simple as in traditional utility offerings and the users can select a computational power “supplier” even for a single application run.

Various virtualization techniques [1], [20], [3] and advances in software package managers have greatly improved deployment of popular software packages or enterprise software. However, if an application must be integrated with target system software, a chosen target is not binary compatible, or virtualization/emulation cannot be used, then the user is nowhere near Utility Computing. Complex software deployment hiders usability of applications and their software conditioning must be maintained by IT experts. The class of applications that are “deployment-hard” is Science and Engineering (SaE) software, which typical targets are computational clusters at high-performance computing (HPC) centers. We characterize SaE applications as sets of cooperating parallel programs with the following attributes: source code availability (open science; often legacy code), nontrivial software dependencies, and target optimization necessities [32]. The advanced nature of these elements foils using SaE applications beyond well-supported environments, such as developers’ or HPC centers platforms where skilled staff handles software deployment. However, as software modeling and simulation have become mainstream tools in SaE, professionals who do not have to be specialists in computer science demand easy access to such tools beyond SaE native platforms.

In addition, the recent surge in on-demand computational offerings encourages professionals to experiment with SaE applications 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, when they want to execute their applications on such targets. Occasionally, SaE software requires porting to enable execution on a particular resource or, reversely, the available code of SaE applications may be tuned to hardware-specific capabilities absent on common platforms. Clearly, more autonomous approach in application deployment and an improved execution model are needed; however, these subjects in the context of SaE applications 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 application is successfully installed. As a result, users may execute their applications 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 SaE software, to single hosts, Grids, and IaaS clouds, to PaaS services. We believe, that smooth switching between computational power providers, even for a single application run, may lay a foundation for the long-standing goal of Computing-as-a-Utility.

II. RELATED CONCEPTS

The excessive effort related to software deployment is a recurring motif in many projects [37]. Typical build automation software (e.g., GNU Make [11], CMake [6], Ant [4], SCons [18], Gradle [12]) 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 the SaE environments. Moreover, locking onto a particular build automation tool may require porting of the existing build solution for some dependencies of a particular SaE application. Another inconvenience related to the use of modern build tools that generate source dependency graphs, such as SCons, may be the time of build for often enormous weight of SaE packages, e.g., Trilinos [31].

We propose a more general approach based on the concept of metabuild that enables all software dependencies in their native deployment formats. Our metabuild does not introduce yet another replacement for build systems but aims at management of existing deployment methods. Although our solution requires a shell and a basic system toolkit as a direct dependency, we may bootstrap extra deployment and software dependencies in the user space.

Popular build tools support probing functionality, which checks presence and appropriateness of requested dependencies, such as specific versions or computation precision. Such tests may become troublesome if a chosen target offers several versions of the dependencies in separate directories or managed by Environment Modules [9]. Moreover, existing tools are capable of merely reporting problems and leave the resolution to the user. As we aspire to provide automatic software provisioning, we need to use probing facilities to detect and find a solution for the issue. In order to fix the problem, our toolkit may switch to another software suite or try another soft-condition method. Diagnostic tests are also important to keep the build environment homogeneous during the entire application deployment, i.e., compatibility of subdependencies. To implement our probing, we intend to use and extend existing mechanisms from tools such as SCons, CMake, or GNU Autoconf [10].

The installation of software packages with automatic dependency resolution is addressed by package management systems, such as RPM [17] or various implementations of Ports [14], [13], [15]. 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 SaE applications 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, SaE build is often proprietary, which may greatly hamper unification in a chosen build format: SaE software often requests specific versions and selective compilation or patching of its dependencies [32]. 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 SaE application deployment toolkit must address extraordinary SaE application build requirements. However, we intend to use target-specific package systems if they are available, as they substantially simplify dependency provisioning.

Executing an application 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 SaE software stack within the image. Projects such as OSCAR [29] or rPath [24] deliver customized OS images with requested software preinstalled from package repositories; such the approach greatly standardizes conditioning of the system software. 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 SaE application using a minimalistic Linux distribution for a specific IaaS platform–SaE application pair.

EasyBuild [27] facilitates installation of SaE applications 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 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 [30], Chef [5], or Sprinkle [19], 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 administrators 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. As an extra feature, even if it breaks the overall portability, CM’s may provide bare command execution and mix command-based operations with other built-in deployment steps. Variety of SaE software deployment issues causes that the command execution capability must be used for SaE 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 SaE applications with the assistance of the adaptive middleware ADAPT (ADaptive Application and Platform Translation). In an overview, ADAPT allows users to select different targets for their application by automatic and transparent software conditioning [22], as it is illustrated in Fig. 1.

ADAPT proposes a simple model of application execution: in order to sustain an application (support the application 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 applications requirements to match to actual resource capabilities. Fig. 2 shows schematically the concept of the application-target adaptation: the same application may require a different set of adapters in order to be executed on a different resource.

Note, that the application requirements remain constant whereas the target capabilities vary from machine to machine. As a result, one target may be ready to build the application instantly, while another target must undergo multi-stage software conditioning in order to meet the application 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 a permanent storage capability on a diskless host). By extending capabilities rather than modifying requirements, our approach performs the bottom-up adaptation.

Another approach is to alter the applications modifying their requirements in order to fit to resource capabilities deviant from originally expected (cf. Fig. 2). For instance, it is possible to automatically translate the applications source code to other programming languages (e.g., Fortran to C [25]) or substitute libraries. In this manner, adapters approach the resource capabilities in the top-down manner. As such modifications may be safe for syntactic mappings, they offer a lesser scope...
of adaptation and the major method to couple applications with targets remains resource provisioning. For these reasons, we delve into soft-conditioning of resources to provide the requested level of capabilities.

In this paper, we focus on provisioning the SaE 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 the deployment automation, the user experiences no operational difference between various targets and views different offerings as the computing utility. Further, this promotes experimentations with SaE applications 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 applications from the SaE class onto a wider range of computational resources. The SaE soft-conditioning is particularly difficult as SaE applications 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 SaE applications beyond on-premises or supercomputer center machines and aims at offering the SaE 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 SaE applications 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 application. Such idea greatly supports the Computing-as-a-Utility vision, helps popularize SaE software applications beyond a close community, and may increase the overall deployment productivity at HPC centers.

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 application–target pair; this concept is presented in Fig. 3. A metadeployment script calls matching deployment recipes retrieved from a recipe repository and defines the correct order of dependency provisions. 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 SaE 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 in the repository to optimize future deployment on the same or similar targets.

As we mentioned in Section II, typically, SaE application 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 applications 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 SaE software deployment.

B. Target Environments

SaE applications 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 administrators 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 cloud offerings can be easily specialized with the use of successive conditioning [35] 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 (i.e., the OS images offered by the cloud provider), we will generate an OS image that (1) is tailored both for the application and the underlying virtualized platform and (2) can be formed even for single execution of a given application.
As such the design may ignore objectives other than the application execution, i.e., we can reduce the system software to bare, essential functionalities that are necessary yet sufficient to sustain the application execution. This significantly reduces both the size of the image and the operating system noise [26]. Thanks to this reduction, we intend to improve execution performance, decrease the image upload and boot time, thus lowering the resource utilization costs. Moreover, assembly and configuration of the image may support abstract, utility-like execution of the applications as, from the user perspective, execution of a program would be as simple as sending an image file to the virtualized target and staging out the results.

C. Operational Scenarios

To deploy an application, 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 Fig.4. 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 standpoint, we plan to study this option more carefully. So far, we have experimented with Tiny Core Linux [7] (a functional OS of size below 8MB) 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 application statically. Consequently, this may limit the binary requirements to syscalls, thus benefiting performance. Finally, proper configuration of the image may cause that the application will start when the instance boots (init) and the instance may be terminated on the application’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.

A. Metadeployment script

SaE 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 application-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 (metadata). An example of a metadeployment script is shown in Listing 1.

Listing 1. A metadeployment script for LifeV simulations [34]

```
#!/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/\(BLASLIBPATH\)/$blas/; s/\(LIFELIBPATH\)/$lib/" < Makefile.in.sample > Makefile.in
# B. build; fix errors if they happen
adapt monitor make
```

B. Recipe design

We design the ADAPT recipes to implement abstract dependencies provisioning for metadeployment scripts. As such a task is internally recursive—a software deployment script calls recipes to enable its requirements—we use the idea of generic metadeployment scripts for soft-conditioning 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 application 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 Fig. 5. 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. In addition, DInterface imposes a life-cycle scheme that must be implemented by recipes. 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 application requires only a differential update; also one recipe may use the content of other recipes.

C. Recipe Repository

Recipe repository 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 main mechanism to classify the recipes. These tags should 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 should describe compatibility of their script with
targets, such as CentOS, version>6.0, x86-64, ksh, or Nvidia-Tesla. The tags may be updated after successful application of a recipe on a given target. In this way, the ADAPT repository can improve its performance by updating a confidence score of the recipe–tag bindings—if the command is reexecuted on the same or similar machine, better fitting recipes will be fetched first.

We decline to predefine those tags, expecting that the semantic structure will emerge when the system is in use (folksonomy [28], machine text classification). Also, more structured classification mechanisms such as RDF taxonomies [16] may be provided. This classification is employed when ADAPT queries for recipes; the deploy command (cf. Listing 1) may be specialized to control which recipes are being retrieved from 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; ?id compatible recipe:23523" [21] for the RDF-based search (note that recipes may be referenced directly). The target related tags are provided implicitly by the adapt command; however, the user can specify another set of target tags to control crossdeployment.

We intend to provide the recipe repository for a wider community of system maintainers and developers. We hope to adopt the Wikipedia model where members of the community create and maintain the content. Also, using tags as the mechanism to describe the recipes, we count on the community to contribute in evaluation of recipe usability by modifying tags (community feedback). From a conceptual vantage point, this idea is similar to question-and-answer webpages such as Stack Overflow but requires more formalized approach. In return, the repository may offer a common, centralized source of working snippets ready to apply on a given target.

D. 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 command. It may require journaling of disc operations, similar to what we studied in [33], 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 be created by the community they may contain malicious snippets or unintentional errors compromising the system stability. Additional mechanisms such as the MD5 checksums for the recipe content may be considered for verification.

VI. Tests

The first applications that we would like to equip with the ADAPT deployment mechanism are LifeV-based hemodynamic simulation applications (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 [34]. 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, Dolt [8], and EasyBuild. We are interested in answering the following research questions: how the SaE application’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. Next, 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 an ADAPT deployment metabuild script for their target. 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. The natural next step is to provide recipes for another SaE with similar to LifeV requirements. Next, we will challenge our approach with VisIt. This test is interesting as the monolithic VisIt build script—a shell-based build with over 15 KLOC—provides an autonomic deployment mechanism. We believe that our approach will provide an equivalent deployment with just several generic ADAPT commands and relevant recipes.

Our experimental development of ADAPT is available at http://code.google.com/p/dadapt (Python, git). In this first phase, we focus on testing the concept of recipes and implementing tag-based queries. Next, we will propose the client-side toolkit (the adapt command), which is the crucial component of metabuild scripts, and experiment with the recipe repository. We plan to deliver these four elements in the first version of ADAPT.

VII. Summary

Our proposal aims at providing ways for automatic software conditioning with the use of generic, application requirement-aware metadeployment scripts.

Application deployment remains challenging, especially if software needs to be constructed from source codes. This becomes often an extremely difficult task in the context of SaE applications that use hybrid programming models, exploit various parallel processing mechanisms, and depend 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 computational center because deployment of an application they are interested in is so challenging that they
cannot afford its deployment on other machines; consequently, they are vulnerable to disadvantages such as low resource availability or excessive levies. These issues may significantly hinder experiments with other machines, offerings, and emerging technologies as well as may obstruct 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 application 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 wide spectrum of machines, from typical SaE machines, such as supercomputers and high-end clusters, to a single workstation and virtualized platforms, such as grids and IaaS clouds. 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 SaE application—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 application–machine pair. Situation-specific images also decrease a number of logical steps required to start an application—just (1) upload the image, then (2) download the results.

Such the “total executability” approach also raises research concerns about a possible range of the application–target mapping; another goal of the ADAPT project is to investigate similarities between different programming models and computational platforms to relax bounds between the applications and their typical targets [36]. For instance, using specialized images for IaaS clouds may transform some IaaS cloud and grid offerings functionally into Platform-as-a-Service resources. The grand research outcome is to support the Computing-as-a-Utility idea—the vision where users may select any computational resource in order to execute their application. This may be realized by providing a tool that automatically and transparently mediates between an application and targets/providers, even for a single run of the application.

From a more abstract point of view, such automatic and transparent execution may occasionally hinder experiments with other machines, offerings, and emerging technologies as well as may obstruct progress in science.

ACKNOWLEDGMENT

Research supported in part by US National Science Foundation Grant OCI-1124418.

REFERENCES