Technical Report

TR-2014-005

Multi-target Application Deployment with Assistance of Automatic Adaptive Metabuild

by

Jaroslaw Slawinski, Vaidy Sunderam

Mathematics and Computer Science

EMORY UNIVERSITY

Multi-target Application Deployment with Assistance of Automatic Adaptive Metabuild

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, machine heterogeneity obstructs multi-target deployment for complex and multi-dependency scientific and engineering codes and makes this aim intractable. We describe a proposal for a metadeployment toolkit, called ADAPT, based on reusable recipes addressing 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

Software modeling and simulation are the basic tools in Science and Engineering (SaE) that help understand the observations as well as predict the evolution of studied phenomena. Along with increase of science knowledge, scientific models become more complex and aggregate more disciplines; as a result, SaE software implementing models are more complex too. As a consequence, sustaining more advanced models requires cutting edge, high performance hardware, experimental system software stacks, and novel programming techniques, which additionally complicate assembly of SaE software. We characterize SaE applications as sets of cooperating parallel programs with the following attributes: source code availability, nontrivial software dependencies, and target optimization necessities [1]. The advanced nature of these issues foil using SaE applications beyond well-supported environments, such as developers' or HPC centers' platforms.

On the other hand, the recent surge in on-demand computational offerings encourages users to experiment with SaE applications on a broader assortment of resources. Despite unavoidable performance degradation, 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). However, to facilitate the execution on other resources, an improved, automatic approach to application deployment is required as end-users cannot rely on user support offered at HPC centers. Occasionally, SaE software requires *porting* to enable execution on heterogeneous resources as the code already can be adapted to specific (atypical) hardware architectures and software stacks; e.g., SaE applications may be tuned to hardware-specific capabilities absent on another platform or execution on new resources may be hindered by target-related soft-dependencies that need to be satisfied at build and launch stages. The subject of providing dependencies for SaE applications in a seamless manner has not attracted sufficient research attention yet and solving dependency related issues usually imposes an unnecessary burden on scientists who may be experts in physics or biology but not necessarily in computer science.

This paper delivers a proposal and a use case of a system that provides automatic, adaptive, and transparent multi-targeted deployment solution. This design employs reusable *deployment recipes* that encapsulate and store expert knowledge related to software conditioning. Proper chaining of these recipes formulate *automatic deployment scripts* that probe and soft-condition the target environment until the considered application is successfully installed. As a result, users may focus on executing their applications on wider range of resources without drudgery related to deployment phases. Our proposal contributes to the usability enhancement of computational offerings and has the potential to increase productivity in HPC by providing systematic and automatic environment conditioning, as well as support know-how sharing beyond a narrow group of specialists. Furthermore, enabling smooth switching between *computational power* providers, even for a single application run, may become a realization of the long-standing goal of Computing-as-a-Utility.

II. Related Concepts and Work

The excessive effort related to software deployment is a recurring motif of research and practical implementations [2]. Typical *build automation software* (e.g., GNU Make [3], CMake [4], Ant [5], SCons [6], Gradle [7]) is designed to build a single, but possibly multiproject, software bundle, such as an individual library or executable. Frequently, such tools address particular programming languages, such as C/C++ or Java, and may depend on uncommon in SaE environment dependencies. Moreover, locking into a particular automation tool requires porting of the current build mechanisms for available SaE

This is an extended version of the paper submitted for 10th International Conference On Parallel Processing And Applied Mathematics.

software components. We propose a more general approach based on the concept of *metabuild* that enables all software dependencies and delivers the SaE application binary for the specific target. Our metabuild does not introduce yet another replacement for build systems but aims at *management* of existing deployment steps. Moreover, our solution requires merely a shell; however, we can bootstrap any extra dependency in the user space. In regards to the internal strategy, our method is similar to CMake: instead of directly handling software components, the toolkit generates a *metadeployment script* for a specific deployment scenario; then the user runs this script on the target.

Popular build tools support *probing* functionality, which check presence and appropriateness (e.g., expected features, such as specific versions or computation precision) of requested dependencies. Such tests may become troublesome if a chosen target offers several versions of the dependencies either in separate directories or switched by Environment Modules [8]. Moreover, existing tools are capable of merely reporting problems and leave the resolution to the user. As we aspire to provide an automatic software provisioning, we need to use probing facilities to detect and find a solution for the issue. To fix the problem, our toolkit switches to another software suite or tries another soft-condition method. Diagnostic tests are also important to keep the build environment homogeneous during the entire application deployment (e.g., compatibility conditions between dependencies). We intend to use and extend the SCons probing mechanisms by implementing functionality of checking tools' properties (such as in GNU Autoconf [9]) and finding solutions in case build issues arise.

The installation of software packages with automatic dependency resolution is addressed by package management systems, such as RPM [10] or different implementation of Ports [11], [12], [13]. Selected software is delivered in a form of standardized packages downloadable from (external) repositories. The package includes the software payload (either precompiled or in sources) 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, the nature of SaE build systems is often proprietary, which greatly hampers conversion to a standard format: SaE software often requests specific versions and selective compilation or patching of its dependencies [1]. The way packages handle automated 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 needs. However, we intend to use target-specific package systems if possible as they substantially simplify dependency provisioning.

Executing an application on IaaS clouds may seem to be a simple task as the user can entirely reshape execution environments of instantiated virtual hosts. However, only a fraction of cloud providers allow users to run a particular OS whereas remaining 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 [14] or rPath [15] deliver OS images with collections of requested software from package repositories, which greatly standardizes conditioning of the system software. However, the resulting image easily becomes overinflated in terms of size and provided services, which may lead to increased upload overhead and service cost. In order to mitigate this problem and improve the computational efficiency, we propose another solution for assembling images for IaaS. Our metabuild composes images for a single run of a SaE application using OS kernels tailored for both a specific IaaS platform and SaE application requirements.

EasyBuild [16] 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 deployment steps, captures actions that users routinely perform to install software. Such mapping is more natural for the users as they may immediately preserve their knowledge related to software installation. As we do not impose any formalism, users may provide any deployment steps without resorting to extending the toolkit. Finally, to logically connect deployment phases, we use deployment variables grouped into interfaces that bear resemblance to SCons construction environments.

Configuration management (CM) tools, such as Puppet [17], Chef [18], or Sprinkle [19], are often used to provide a well defined software set on multifarious targets and support variety of OS, also the Windows family, in a transparent way. Such tools are beneficial for the system maintainers who can automatically and in a repeatable way deploy software on remote hosts inside their administrative domain. For these reasons, CM tools are frequently recommended to the users by IaaS cloud providers if their offerings do not accept customized images. The main strength of that projects is that the user may define the software stack in a semi-declarative way using predefined actions and properties. However, CM's also may provide some bare command execution and mix command-based operations with other built-in deployment procedures. As SaE software deployment must be supported by the command execution capability to handle atypical software needs, CM tools with command launch properties would be considered as a tool for SaE application conditioning. How-

ever, we claim that to provide a usable tool for SaE installation, the command execution and standard output processing aspects must be elevated to key components of the solution. Also, it is interested how Domain-Specific Languages may address specifics of the software provisioning [20] as we also want to use a high level programming language as the interface to provide deployment descriptors.

Another popular solution to provide a concise set of software dependencies and to build the whole project in a homogeneous environment is to provide a single build script, e.g., in shell, that deploys the entire software stack. Such approach is used to build VisIt [21], a SaE class visualization toolkit. Disadvantages of this method are an unmanageable build script, hard-listed dependencies, and time-consuming execution (sequential execution). We may improve this deployment by providing a method to generate such monolithic build scripts adapted to a particular target using, e.g., already available software, if compatible, switch between multiple versions of dependencies, and deploy separate dependencies in parallel, which may reduce the overall deployment time.

III. ADAPT Project

The primary goal of this project is to extend the usability of hardware architectures by enabling execution of chosen, *unmodified* SaE applications with the assistance of the *adaptive middleware environment* ADAPT (ADaptive Application and Platform Translation). ADAPT allows users to select different targets for their application and adapts resources to applications' requirements [22] , as it is shown in Fig. 1.

ADAPT proposes a simple model of 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., routines stored in dynamic libraries), binary compatibility, typical communication or interaction interfaces, etc. The resource capabilities are all facilities offered by the resource and exposed by its system software such as storage (e.g., local file system), inter-process communication (e.g., present network fabric), 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 adapters for the application. Fig. 2 shows schematically the concept of the application–target adaptation: the same application may require a different set, approach, and range of adapters in order to be executed on a different resource.

Note, that the application requirements stay constant whereas the target capabilities changes 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 our opinion, 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., a virtual machine, dynamic binary translation, emulation) or outsourced (e.g., providing a permanent storage capability on a diskless host). In this way, adapters approach the application requirements in the bottom-up manner.

It is also possible to transform the applications so they shift their requirements (cf. Fig. 2). For instance, it is possible to automatically translate the applications source code to another programming languages (e.g., Fortran to C [23]) or substitute libraries calls; In this way, adapters approach the resource capabilities in the top-down manner. As such modifications are limited to syntactic mappings, they cannot significantly influence the adapters and the major method to match applications with targets remain resource provisioning. For these reasons, we focus entirely on soft-conditioning of the resource to the requested capability level.

In this examination of the ADAPT idea, 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 paper. We believe that deployment should be fully automatic to promote experimentations with SaE applications on targets differing from well-supported, typical execution platforms. In this case, ADAPT applies software components on the resource, layer by layer, until the requested level of specialization is achieved.

The previous experiments with ADAPT and current project are depicted in Fig. 3—we studied related issues and this proposal extends our research: we (1) analyzed a wide set of HPC production codes to understand commonalities related to their build and proposed profiles that encapsulate and enable processing of build-related knowledge [1], (2) studied transformations of less specialized resources into more specialized ones (e.g., a set of ssh-able hosts into a High-Availability (HA) cluster) [24], and, in more extreme adaptation cases, (3) provided methods to reconcile different programming/platform paradigms to enable execution of unmodified applications (e.g., using MapReduce platforms for execution MPI applications) [25].

The current research project is *pragmatic* as the deployment phase must always proceeds any software using. For this reason, we discontinue investigating the extreme cases of adaptation as they are limited to a narrow subclass of applications and targets and impractical from the production perspective (e.g., MPI communication speed implemented on top of a MapReduce's distributed file system is many times slower). Also, we do not want to address PaaS's as they represent usually more specialized targets and often require semantic modifications in the application codes. However, some PaaS services may be taken into consideration as they exhibit a similar to classic resources bare execution model (e.g., MS Azure is capable to execute recompiled, potentially unmodified, MPI applications).

IV. Metabuild Design

In this paper, we propose a design of a toolkit to enhance deployment of applications from the SaE class onto a wider range of computational resources. The SaE soft-conditioning is

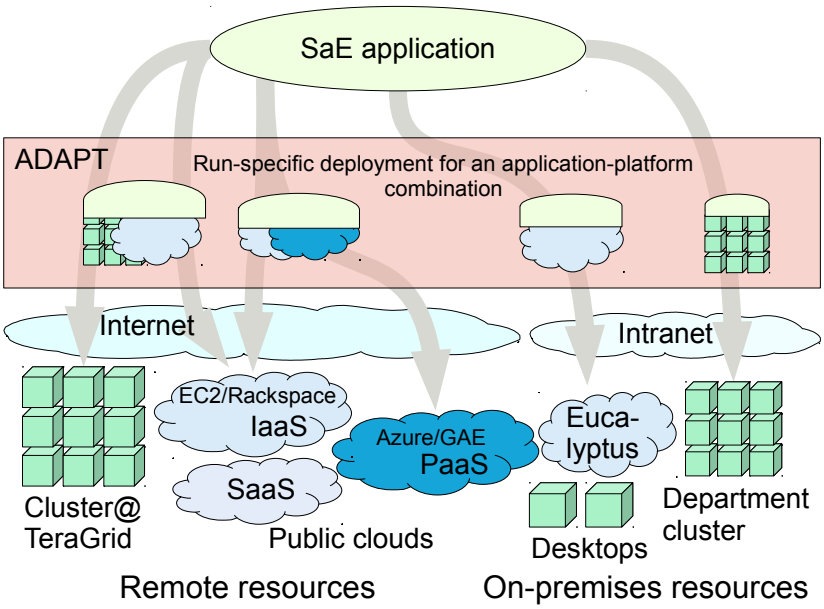

Fig. 1. An application may be executed on different targets thanks to the middleware that generates the situation specific adaptation.

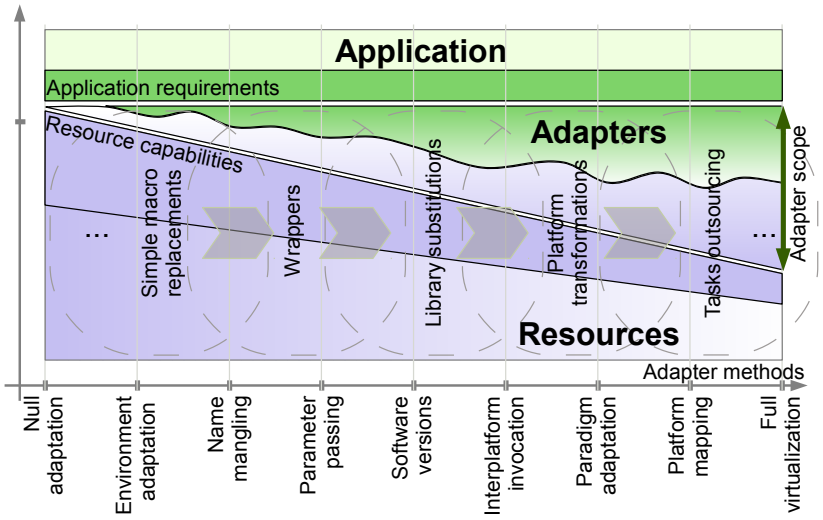

Fig. 2. Adapters must address differences between the constant application requirements and changing from resource to resource capabilities.

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. We propose an enhancement of the usability of the applications beyond on-premises, supercomputer center machines and aim 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 for availability or financial reasons, even for a single run of an application. Such idea greatly support the Computing-as-a-Utility vision, helps popularize SaE software applications beyond a close community, and increases the overall deployment productivity.

A. Target Environment

As we focus on classical computational resources, viz. workstations, clusters, and grids, the level of access privilege to the system shapes available interaction scenarios that users can perform on the machine. Typically, the users have limited access to computers that can be used for SaE applications and perform software conditioning in the *user space*, with all

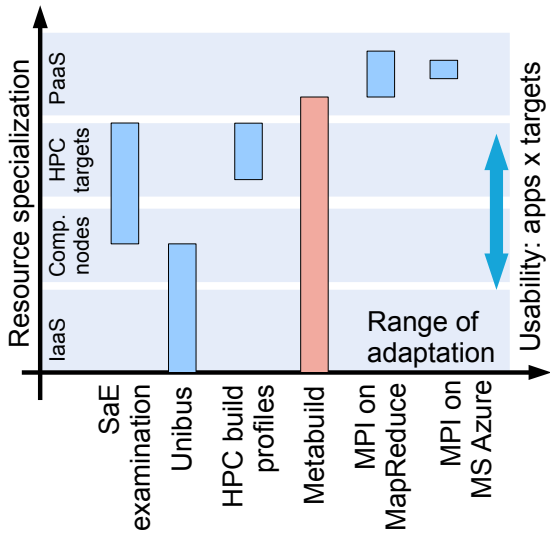

Fig. 3. Usability of our previous and current research projects. "Usability" defines how much applications and targets may be addressed by the given approach. Automatic deployment is the most usable method.

the innate limitations, or ask the admins for support. As our method needs to be as transparent and universal as possible, we assume only user space deployment; however, this does not exclude work in the elevated privileged modes, if possible.

The infrastructure cloud offerings can be easily specialized with the use of *successive conditioning* [26] that yields chunks of classical resources on demand so the software provisioning may be easily performed as for classical resources (or even easier as the users have the privilege access to the virtual instances). However, as several IaaS cloud and grid providers allow the user to provide an operating system, which gives superior flexibility even in comparison to privileged access, we offer a more specialized approach. Instead of applying software conditioning on virtual resources running a standard OS, we will generate an OS image customized both for the application and the underlying virtualized platform. As such image is formed just for execution of the specific application, we can reduce the system software to bare, essential functionality that just sustains the execution, which significantly reduce both the *size* of the image and the *operating system noise* [27]. Thanks to such reduction, we hope to improve execution performance, decrease the upload and boot time as well as lower the cost of using the service.

B. Proposed Deployment

The application deployment process is unavoidable step toward the execution of an application. Often cumbersome, unstructured, and frustrating, a deployment phase is critical for correctness and performance of the execution. Typically, SaE applications are distributed with a build system (e.g., GNU make, CMake, shell), build parameters, and a list of prerequisites such as a dependency inventory, required environment variables, etc. We do not aim at providing yet another build system; rather we intend to *chain* currently separated steps, provide and automatically apply *solutions* for the common problems (i.e., errors during the deployment) as well as make this effort *reusable* and *self-documenting*. Automatization of the deployment is the key factor for enhanced usability; however, this process, especially for complex and not well documented SaE applications, cannot avoid human intervention to solve problems when they appear (e.g., lack of dependencies, incorrect or incompatible versions, compilation errors). For this reason we envisage also a guided, *semi-automatic* mode of the metabuild, during which our tool can acquire from the operator build knowledge and store it for reuse.

C. Operational Scenarios

The ADAPT idea is to generate a *deployment script* for a particular *deployment scenario* defined as the specific application–target pair. The deployment script contains a set of matching *deployment recipes* retrieved from *recipe repositories*; defines the correct order of dependency provisioning and has no own explicit software dependencies (a deployment script depends only on shell and may bootstraps own dependencies, if needed). Next, the users transfer the script to the target and execute it in order to deploy the desired SaE software stack. This script checks and delivers all dependencies by applying, in a designed order, recipes. In case of errors, the script automatically rollbacks and retries with alternative compatible recipe. On success, the chain of steps is saved in the repository as a new recipe for this target, which allows efficiently repeating the deployment for a similar deployment scenario.

To address resources that allow custom images (some IaaS clouds and grids), we will proceed with a similar deployment script but, instead of launching the script on the target directly, we will perform "cross-deployment" resulting in creation of an OS image. We have experimented with Tiny Core Linux [28] (a complete OS of size below 8MB) that we remaster to meet specifics of a virtualized platform and tuned to achieve high computational performance goals (e.g., swap memory switched off). To avoid a superfluous library payload, we intend to build an application *statically* and place its binaries in the image. In this way, we may limit the binary requirements to syscalls, which might further benefit the performance. When the image is used to instantiate a virtual host, the `init` process launches directly the application; the application conclusion may terminate the instance. In addition, the user may request extra services, such as `sshd`, to supplement the execution with monitoring or control. For virtualized resources not accepting a customized image, the users must obtain standard resource equivalents, (instantiate virtual hosts and specialize them to provide, e.g., a cluster-like machine) then apply the generated deployment script. The general operational scenario is depicted in the Fig.4.

V. Metabuild Design

Fig. 5 presents how a use case specific metabuild script is generated from a recipe repository. The users (1) specify the

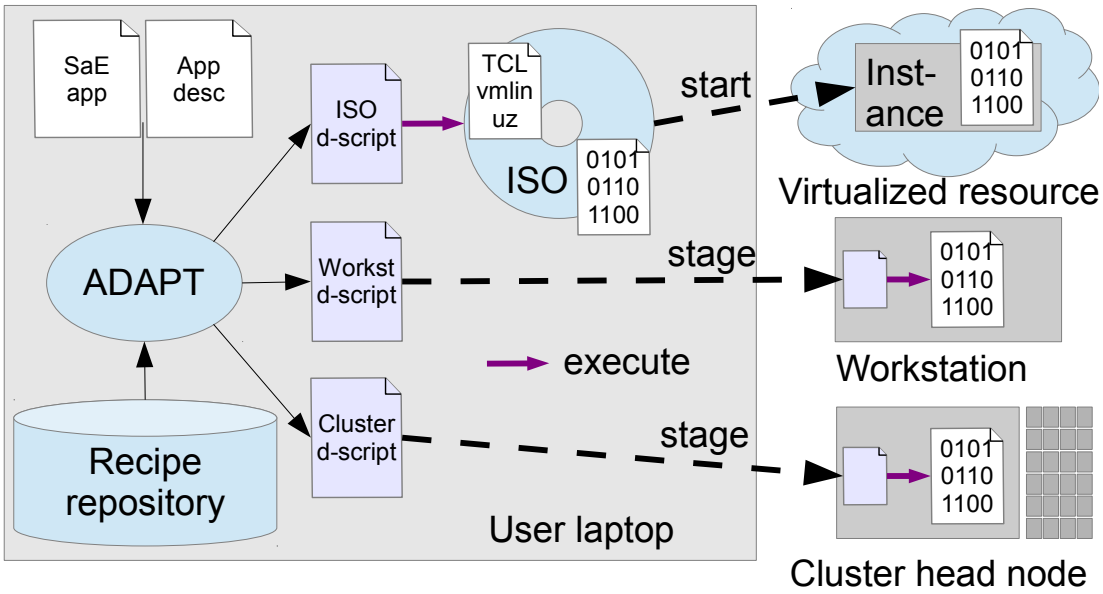

Fig. 4. Users may generate different deployment scripts for different resources. Next, the users execute the script on a given target. For some IaaS clouds, the user may execute the deployment script on an on-premise machine in order to assembly the application–targets specific OS image.

dependency description for their SaE application, (2) prepare the deployment recipe(s) for it, (3) determine the situation specific configuration (a description for the current target; optional parameters for the build), and (4) use the toolkit to generate the metabuild script. Finally, the users (5) execute the final deployment script on the selected target and the script (6) generates the application executables. The next subsections provide details on the following ADAPT concepts: (1) the recipes and (2) their selection, (3) generation of the metabuild script, and (4) the metabuild script execution.

A. Recipes

The main concept in the recipe design is the *deployment interface* (d-interface) that *abstractly* defines a software component to deploy and *formally* defines its requirements and capabilities. The d-interface requirements reference other d-interfaces whereas its capabilities define a unique set of *deployment variables* that represent settings and describe deployment deliverables (e.g., paths to libraries, include files, target specifics). D-interfaces and their capabilities are uniquely identified and semantically well-defined. Also, d-interfaces may extend each other and, in this way, they may provide a higher dependency abstraction (e.g., OpenMPI [29] extends MPI, ATLAS [30] extends LAPACK). In addition, d-interface requirements may provide extra conditions for their dependencies such as a required version, compatibility with a class of targets or with a specific target. In order to deploy a d-interface, there must be at least one deployment recipe that *implements* this d-interface.

The main characteristic of recipes is that they define *actions* that are active processes of deployment. The action may be specified as a shell or Python snippet that performs any tasks enabling given capabilities. Such tasks may include, e.g., searching for specific files in predefined directories, invoking a package management system, loading an environment module, or downloading, building, and installing a software component. As a result of actions, the recipe sets deployment variables defined for its d-interface. The recipe may provide additional functionalities such as probing for specific properties (usability of libraries, functions in libraries, features of compilers, etc.), declaring compatibility with a target or specific dependency version, etc. We also envision additional, nonfunctional parameters that are *estimators* that subjectively characterize recipe properties such as expected performance of to-be-deployed software (performance of libraries tuned for the target is superior to a generic binary) or expected deployment time (compilation may last hours when a precompiled version is immediate). The estimators play an important role in assemblage of metadeployment scripts.

B. Metabuild Script Generation

In order to use our toolkit, first, users provide a description of the SaE application dependencies and deployment activities expressed in terms of recipes collected in the repository. This software description includes a d-interface itemizing a set of requirements and at least one recipe implementing that d-interface. The users also have to create other, missing or undefined application dependency descriptions.

Next, the ADAPT toolkit is used to generate a metabuild script. The users may specify input parameters; the most important parameters set the deployment scenario: (1) ID of the d-interface representing the software to deploy and (2) ID of the target. Extra parameters may specify a version of the software or deployment options establishing a sequence of recipes usage. Based on these parameters, the ADAPT toolkit queries the recipe repository and selects recipes that cover the

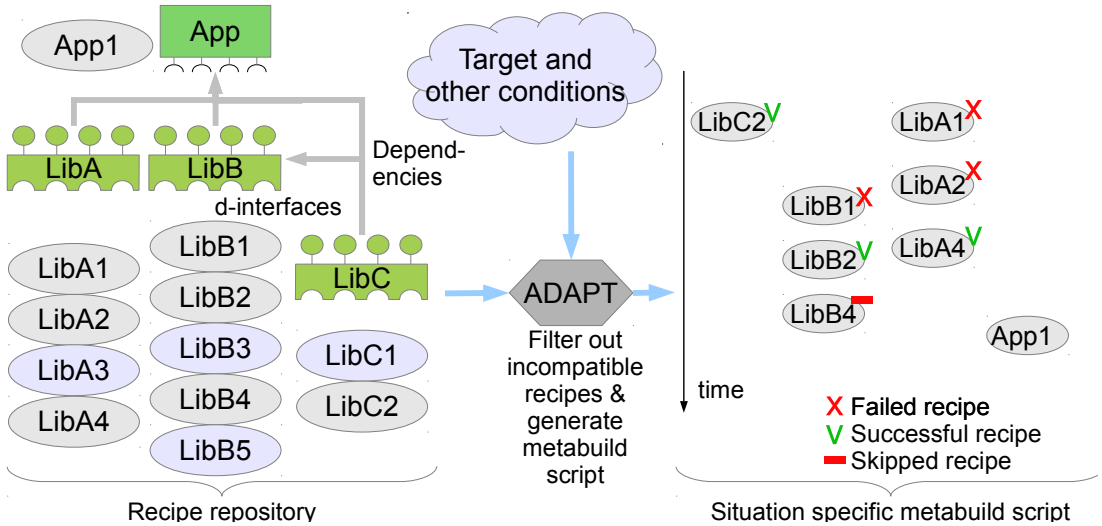

Fig. 5. Metabuild generation and execution: (1) the application and its deployment are described in terms of the recipes collected in the repository, (2) the user parametrizes generation of the deployment script, and executes the script, (4) the deployment script executes the deployment recipes in parallel and manages errors by applying alternative recipes.

dependency graph of the software to deploy and meet additional conditions, such as compatibility with the given target. The toolkit transforms each selected recipe into an executable unit (Python script) that may be run by the metabuild script during the deployment. It is desirable that the toolkit yields more than one matching recipe for a single d-interface—our approach provides automatic problem resolution by applying alternative deployment actions.

The efficient query mechanism is extremely useful to generate correct and compact (a small set of selected recipes) metabuild scripts. We envision that the users may define taxonomies of concepts (e.g., defining targets) in Resource Description Framework (RDF) [31] and classify recipes using these definitions (a recipe may be tagged as, e.g., compatible with a `cluster`, an `Amazon EC2` instance type, or just a specific host). During the recipe selection process, the toolkit employs SPARQL Protocol and RDF Query Language (SPARQL) [32] to select recipes based on the users requirements.

C. Metabuild Script Execution

In the final step of deployment, the users transfer the generated metabuild script into the selected target and launch it. Our goal is to minimize software dependencies for a metabuild script so the script has to be self-sustained and depends only on the basic execution capability of the system; missing dependencies, such as Python, must be bootstrapped. For this reason, a possible format of metabuild scripts is binary or shell script (we assume shell as a none dependency). The metabuild script is aware of the dependency graph for the software to be deployed (Directed Acyclic Graph, or DAG) as well as it should contain all selected recipes. In order to implement the deployment scripts, we intend to use `doit`—a task execution automatization project [33]. Each `doit` script, which is a valid Python module, contains a set of objects that *declaratively* define tasks driven by actions (shell or Python codes); the script identifies dependencies (resources or another tasks) and deliverables (files exposed after the task execution). The `doit` execution computes dependencies among tasks and activates them in accordance with a logical succession. `doit` offers parallel execution and caches task results skipping replicated actions to improve performance of task execution. Our recipes correspond to the `doit` tasks and are inspired by this project ideas; consequently, we can easily transform the recipe actions into the task actions and map other recipe properties into task properties.

As a single d-interface may possess several selected recipes compatible with a target, a proper execution mechanism is needed to choose the recipe that actually deliver the dependency. Recipes contain estimators that assess recipes (optimization of deliverables, expected deployment time, etc.). During the deployment phase, these values serve as *order fields*—when the metabuild script is running, recipes are *tried* in this order. If a recipe action fails, the next recipe in the row is started; if the recipe action succeeds, the remaining recipes are skipped. Using such "recipe redundancy," we adapt to the specific deployment scenario by first applying a recipe that fits best. Such a behavior requires a modification of `doit` to allow executing alternative tasks. Also, as some recipes may change the state of the system, we may add a rollback mechanism to *sandbox* the recipe actions. To do so, we may apply mechanisms described in our previous work [34] to monitor system resources and restore them if needed.

VI. Deployment Repository Example

The first applications that we would like to equipped 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 LifeV programs and their scientific software dependencies are given in [35]. We believe that our approach will deliver an easy deployment method in a form of portable scripts for a variety of targets and popularize our simulation software beyond our local scientific community. A fragment of a recipe repository for a LifeV application is shown in Listing 1.

Repositories are valid Python modules—this greatly simplifies repository maintenance by applying syntactic rules to validate dependencies–d-interfaces bounds and eliminates any description parsing. Actions are a list of shell or Python activities that reflect the casual deployment steps leading to software installation. Extra recipe requirement properties, such as `file_req`, force applying other recipes before applying the current one (i.e., d-interface recipes that deliver the source code of the application may search local directories to fine the source code, download and unpack a compressed file, or clone the sources from a repository). Recipes may contain various tests on the deploying content and may restrict the dependencies by applying filters, such as querying for the content (e.g., checking for a library or function) or functionality (e.g., checking if the C++ compiler supports specific features, such as templates). In addition, we intend to use class inheritance between recipes to simplify creation of similar descriptions. The next elements in repository files are RDF taxonomy entries mapped onto Python classes to simplify the recipe definitions (e.g., for the `targets` attribute, `TARGET` is the namespace of a targets definition file and `workstation` is a specific target concept). Finally, we plan to provide a utility library providing common functions, definitions, and tests to simplify recipes creation.

VII. Summary and Future Work

Application deployment remains a challenging process, especially, if software needs to be constructed from sources. 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 the problem intractable. In this paper, 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 provide a method to *capture* diverse user activities leading to an installation of a software component on a given platform and, next, to *process* and *reuse* this 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. Such an approach has also a research consequence: our ongoing project ADAPT investigates similarities between different programming models and computational platforms to relax bounds between the applications and their typical targets [25]. For instance, using specialized images for IaaS clouds transforms 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.

As future work, first, we want to compare different deployment implementations for our hemodynamic simulations using different frameworks, including Chef, Sprinkle, bare DoIt, and EasyBuild. We are interested in the following issues: how the SaE application's atypical requirements can be expressed in different frameworks, how flexible is the solution as well as what are user effort and reusability of already stored recipes. 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 generate and download a specific deployment metabuild script for their target from our web page. As the next deployment exercise, we will apply our approach to VisIt (a shell-based build with over 15 KLOC). We plan to offer a generator of a target-specific metabuild script—an equivalent of the current VisIt deployment script. However, we expect to improve the present deployment time as our solution may enable independent software dependencies in parallel. Also, we will study the application-oriented image-based deployment for virtualized resources and compare this with the deployment for PaaS platforms.

Acknowledgment

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

References

[1] Magdalena Slawinska, Jaroslaw Slawinski, and Vaidy Sunderam. Portable Builds of HPC Applications on Diverse Target Platforms. In *Parallel & Distributed Processing (IPDPS). International Symposium on*. IEEE, 2009.
[2] G. Wilson, DA Aruliah, C.T. Brown, N.P.C. Hong, M. Davis, R.T. Guy, et al. Best Practices for Scientific Computing. *arXiv:1210.0530*, 2012.
[3] The GNU Make Project. http://www.gnu.org/software/make, 2013.
[4] The CMake Project. http://www.cmake.org, 2013.
[5] The Apache Ant Project. http://ant.apache.org, 2013.
[6] The SCons Project. http://www.scons.org, 2013.
[7] The Gradle Project. http://www.gradle.org, 2013.
[8] The Environment Modules Project. http://modules.sourceforge.net, 2013.
[9] The GNU Autoconf Project. http://www.gnu.org/software/autoconf, 2013.
[10] The RPM Package Manager Project. http://rpm.org, 2013.
[11] The OpenBSD Port Project. http://openbsd.org/porting.html, 2013.
[12] The MacPorts Project Official Homepage. http://www.macports.org, 2013.
[13] The pkgsrc Project. http://www.pkgsrc.org, 2013.

Listing 1. A fragment of a recipe repository for a LifeV simulation

```
# create a d-interface for the application to deploy, declare requirements
class MY_SIM(D_INTERFACE):
  # sim app. may have just one direct dependency, but force the version
  requirements = LIFEV(version('>2.2'))

# declare a deployment recipe for the application in question
# this sim. build recipe is based on Makefile that includes Makefile.in
# defining paths to all application dependencies and required libraries
class my_sim_recipe(RECIPE):
  # bind this recipe with its d-interface
  implements = MY_SIM
  # Makefile.SAMPLE.in is a template for the build, all paths and libraries
  # must be concertized before the make command. We use deployment variables.
  # Note the indirect dependency use---we do not want 'any' Trilinos library
  # but the one that was used for deploying LifeV
  # Dep. variables: dirs, libraries, include files, and other resources
  rep_dict = {'LIFEPATH_HERE': LIFEV.LIB_DIR,
              '$(TRILPATH)/include': LIFEV.TRILINOS.INC_DIR,
              '-lzoltan': LIFEV.TRILINOS.ZOLTAN.PATH, ...}
  # Makefile.SAMPLE.in -> Makefile.in generator
  def gen_Makefile():
    template = open('Makefile.SAMPLE.in').read()
    for k, v in rep_dict.iteritems(): template.replace(k, v)
    open('Makefile.in', 'w').write(template) #now, Makefile.in with real paths

  # another task must deliver the sources (local, wget, or repos.)
  file_req = 'simu/' # provided by a resource delivery recipe
  actions = sh('cd simulation'), gen_Makefile, sh('make -j 4')
  post_check = sh('ls simulation') # any acceptance tests
  ...
class my_sim_recipe_no_mpi(my_sim_recipe): # modify main recipe for a target
  targets = TARGET.workstation;  ...

class simulation_sources(RESOURCE):
  provides = 'simu/'

class simulation_src_local(simulation_sources):
  pre_check = sh('find ~ -type d -name simulation | head -1')
  actions = sh('ln -s ${pre_check.output} simul')
  deployment_time = 1 # estimation: this is fast to provide

class simulation_src_remote(simulation_sources):
  actions = sh('curl http://www.lifev.org/down/sim.tar.gz'),
    py(unpack('sim.tar.gz', 'sim')) # for a python method with arguments
    deployment_time = 2 # estimation: this is slower to provide

class LIFEV(D_INTERFACE): #reusable LIFEV deployment description
  capabilities = LIB_DIR, INC_DIR  # it may provide each lib. as sep. path
  requirements = TRILINOS(lib('ml')), MPI, CPP(TEST_TEMPLATE) # condit. req.
```

[14] Chokchai Box Leangsuksun, Lixin Shen, Tong Liu, and Stephen L. Scott. Achieving High Availability and Performance Computing with an HA-OSCAR Cluster. *Future Generation Computer Systems*, 21(4), 2005.

[15] J. Craig. Cloud Coalition: rPath, newScale, and Eucalyptus Systems Partner on Self-Service Public and Private Cloud. *Enterprise Management Associates*, 2010.

[16] Kenneth Hoste, Jens Timmerman, Andy Georges, and Stijn De Weirdt. Easybuild: Building software with ease. *High Performance Computing, Networking Storage and Analysis, SC Companion:*, pages 572–582, 2012.

[17] James Loope. *Managing Infrastructure with Puppet*. O'Reilly Media, Inc., 2011.

[18] The Chef Project. http://www.opscode.com/chef, 2013.

[19] The Sprinkle Project. https://github.com/sprinkle-tool/sprinkle, 2013.

[20] Sebastian Günther, Maximilian Haupt, and Matthias Splieth. Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of It Infrastructures. Technical report, Technical report, Very Large Business Applications Lab Magdeburg, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2010.

[21] The VisIt Project. https://wci.llnl.gov/codes/visit, 2013.

[22] Julien Bourgeois, Vaidy Sunderam, Jaroslaw Slawinski, et al. Extending executability of applications on varied target platforms. In *High Performance Computing and Communications (HPCC), 13th International Conference on*. IEEE, 2011.

[23] Stuart I Feldman. A Fortran to C Converter. In *ACM SIGPLAN Fortran Forum*, volume 9, pages 21–22. ACM, 1990.

[24] Jaroslaw Slawinski, Magdalena Slawinska, and Vaidy Sunderam. Unibus-managed Execution of Scientific Applications on Aggregated Clouds. In *Cluster Computing and the Grid, International Symposium on*. IEEE/ACM, 2010.

[25] Jaroslaw Slawinski and Vaidy Sunderam. Adapting MPI to MapReduce PaaS Clouds: An Experiment in Cross-Paradigm Execution. In *Utility and Cloud Computing (UCC), 5th International Conference on*. IEEE, 2012.

[26] Jaroslaw Slawinski, Magdalena Slawinska, and Vaidy 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] Kurt B. Ferreira, Patrick Bridges, and Ron 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.

[28] The Core Project. http://tinycorelinux.net, 2013.

[29] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[30] R Clint Whaley and Jack J Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27. IEEE Computer Society, 1998.

[31] The Resource Description Framework Project. http://www.w3.org/RDF, 2013.

[32] W3C Recommendation: SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query, 2013.

[33] The DoIt Automation Tool Project. http://pydoit.org/, 2013.

[34] Magdalena Slawinska, Jaroslaw Slawinski, and Vaidy 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.

[35] Jaroslaw Slawinski, Tiziano Passerini, Umberto Villa, Alessandro Veneziani, and Vaidy Sunderam. Experiences with Target-Platform Heterogeneity in Clouds, Grids, and On-Premises Resources. In *26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. IEEE, 2012.