The Harness Workbench: Unified and Adaptive Access to Diverse HPC Platforms

U.S. Department of Energy Grant DE-FG02-06ER25729
Final Report

Vaidy Sunderam
Department of Mathematics and Computer Science
Emory University, Atlanta, GA 30322, USA
vss@emory.edu

1 Overview

The primary goals of the Harness WorkBench (HWB) project were to investigate innovative software environments to enhance the overall productivity of applications science on diverse HPC platforms. Two complementary toolkits were designed and developed: one, a *virtualized command toolkit* (VCT) for application building, deployment, and execution, that provides a common view across diverse HPC systems, in particular the DOE leadership computing platforms (Cray, IBM, SGI, and clusters); and two, a *unified runtime environment* that consolidates access to runtime services via an adaptive framework for execution-time and post processing activities. This project is a joint effort between Emory University, Oak Ridge National Laboratory (ORNL), and the University of Tennessee. The three institutions involved in this project worked closely together; our project methodology was driven by regular meetings of all personnel, where technical design, operational modes, compatibility aspects, and integration issues are discussed in depth, for completed and ongoing work as well as for planned efforts at each individual site. The subprojects undertaken at each location are linked and address complementary aspects of the project; this report describes work done at Emory University.

2 Overall progress

The Emory focus within the overall project has been on the VCT that introduces an abstraction layer between the user and platform-specific compilers, linkers, MPI implementations, queuing systems, and other components. Our research is aimed at unifying access to analogous services available on different platforms, and allowing pre-configuration at the system level and at the project level. The project concentrated on: (1) formulating and conceptualizing the VCT toolkit; (2) in-depth analysis of resource capabilities in order to develop appropriate description mechanisms; and (3) a study of portability of HPC applications across heterogeneous computing platforms. We developed the Zero-Force MPI (ZF-MPI) toolkit for automating build, installation, run, and post-processing stages of MPI applications across heterogeneous architectures. We also experimented with support for porting applications and prototyped an Eclipse plugin called Portlug-in.

In the later project phases we refined (based on experiences gained from the previous period) the VCT concept that resulted in a new architecture of the toolkit, now called the *Harness Workbench Toolkit* (HWT)

that embraces portability issues, build, and execution aspects related to HPC applications. Further, we designed a source code conversion module to address routine porting activities, and experimentally verified this on production molecular dynamics codes (CPMD). With regard to the build phase we developed and experimentally tested on the CPMD code a *late binding* mechanism that enables dynamic concretization of target platform specific information stored in profiles. In parallel, we developed a preliminary HWT prototype which, in its current version, manifests as an Eclipse plugin. In order to verify the viability of our approach we demonstrated the HWT prototype to application scientists and experts at ORNL, namely Pratul Agarwal, Mark Fahey, Arnold Tharrington, and Ricky Kendall. Pratul Agarwal is an application scientist who works on multi-scale modeling of various biomolecular complexes. Mark Fahey and Arnold Tharrington are computational scientists who work in the scientific computing group headed by Ricky Kendall.

In the final stages of the project, we focused on the deployment phase of high-end computing applications where target environments are prepared, executables and data files staged, and execution runtime environment parameters are set. Our work is manifested in a tool called Unibus, aimed at facilitating provisioning and aggregation of multifaceted resources from resource providers and end-users perspectives. To achieve this, Unibus proposes (1) the Capability Model and mediators (resource drivers) to virtualize access to diverse resources, and (2) soft and successive conditioning to enable automatic and user-transparent resource provisioning. We have tested a prototype implementation of Unibus on high-end computers, grid systems, and clouds.

2.1 HWT architecture

In order to address the porting issue that is very important in the build process, we enhanced the Harness Workbench Toolkit (HWT). The HWT architecture shown in Figure 1 consists of three pluggable layers, guided by situation-specific profiles, that reflect the developer's activities related to source code adaptation, the actual build process, and execution. The HWT facilitates code adaptation through porting assistants

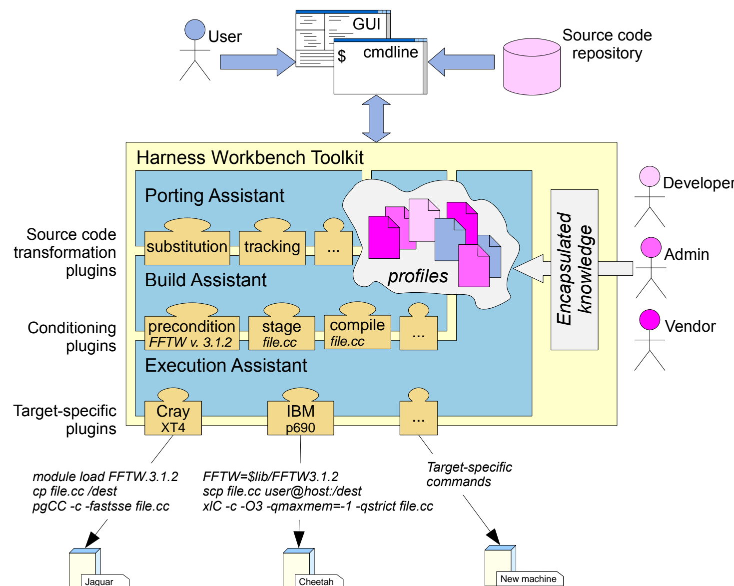

Figure 1: The Harness Workbench Toolkit architecture

that support end-users by identifying snippets that need to be converted, suggestions of conversions that

are known to be safe (e.g., POSIX function substitutions), and highlighting areas where manual code modification might be beneficial, e.g., manual loop unrolling. The build-assistant comprises plugins that are responsible for preconditioning, staging, and compilation. Application launch activities are performed by the execution layer that generates and executes target-specific commands. The build and execution layers are inherited from the original VCT architecture and we utilize the VCT name while referring to those two layers. Toolkit behavior is configurable and tunable through declarative profiles that incorporate target-specific knowledge. Profiles may be shared, reused and created by vendors, site administrators, developers or users. They contain information at the system (software and hardware), application, and user levels and include such details as available compilers and their flags, library paths, or environment settings. The end-user interacts with the HWT by issuing generic commands that are processed by all or selected layers. For example, execution of a generic run command may involve, depending on the actual context, processing by all three layers or only by the execution layer (e.g., if an executable file is already present on the target platform).

2.2 HWT prototype

In Fall 2007, we presented to ORNL developers, the initial HWT prototype with the following functionality: (1) pluggable graphical interface with the ability to define and execute simple scripts to facilitate common build tasks; (2) the "project" model as implemented in HWT; (3) a convenient subsystem for Fortran source code modification; (4) support for repositories (SVN and CVS); (5) a supporting SSH terminal for specific shell tasks; (6) mechanisms to exploit SSH channels (sftp, exec, shell) within one authenticated SSH session; (7) a convenient method for synchronizing files between the local and the remote site. The HWT promotes a shift in the interaction model between the end-user and the computing system. Instead of working directly on a remote front-end node, the end-user works locally through a unified interface to platform-specific tools.

The toolkit was well received by ORNL developers. As a practical outcome, we acquired a few useful suggestions for the HWT that we plan to implement soon. After obtaining access to ORNL machines at the end of November, we performed a successful experiment that aimed to build and execute the CPMD code on Jaguar by using the HWT prototype. This exercise helped us practically test the functionality provided by the HWT. Based on initial feedback from ORNL developers in regard to the HWT prototype demonstration and the CPMD build experiment in a production environment, we identified one main issue (file synchronization) and developed a new HWT prototype version that provides the same functionality but with the improved, according to ORNL scientists' suggestions, synchronization facility. Figure 2 presents the current HWT graphical environment in a typical working layout. The left pane in Figure 2 represents files on the local site that belong to the given project. The rectangular icon decorators (green or gray) indicate whether the file is under synchronization (green) or not (gray). In order to obtain a comprehensive view on both the local and remote sites, the end-user may use the synchronization view presented in the right bottom pane in Figure 2. This visual information is especially convenient for end-users to quickly spot conflicted or obsolete files when modification of the build files on the remote site (via the HWT terminal or external ssh connections) is taken into account. The right upper pane presents a typical editor through which the user can locally modify necessary build files, save them, and then automatically transfer them to the remote site via the synchronization mechanism. We note that this subsystem exploits and takes advantage of the experience gained during developing ZF-MPI in the previous project period in regard to file synchronization (we use the same SSH library implementation, namely *jsch*).

In addition to the development of the integrated graphical toolkit, we also performed two separate experiments related to the (1) porting and (2) build and execution layers.

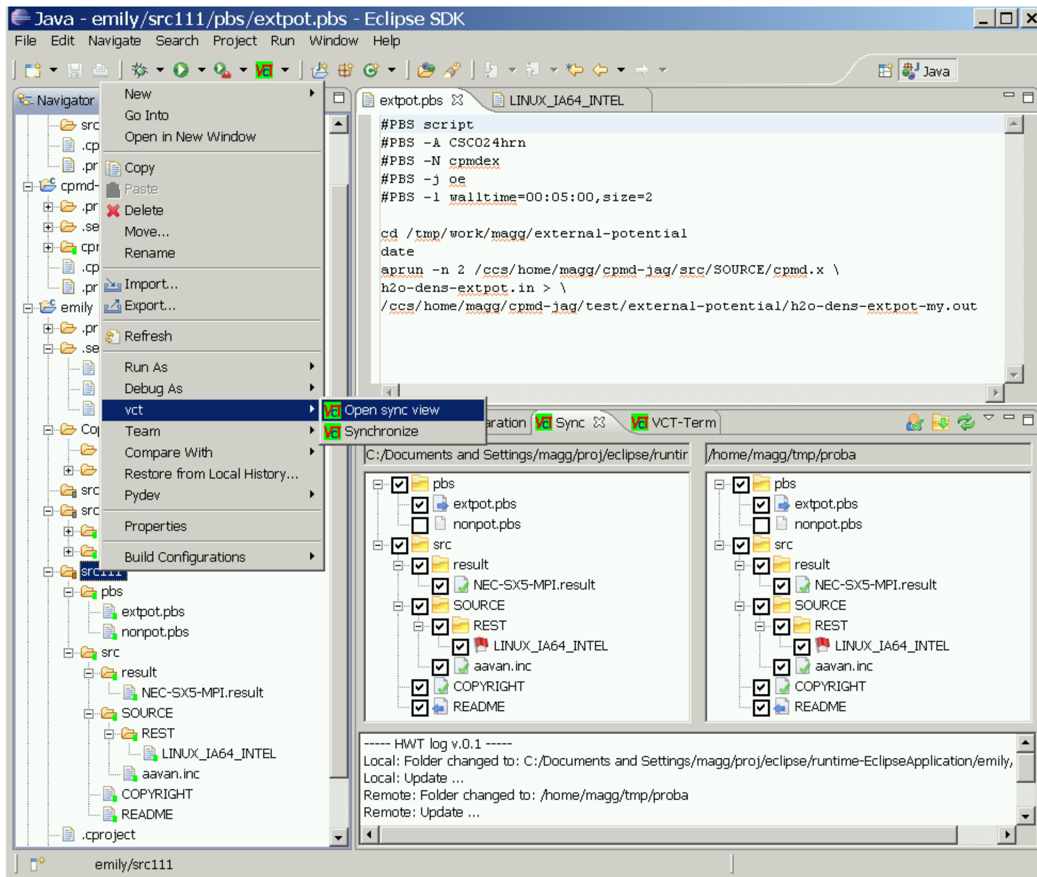

Figure 2: The HWT graphical environment

2.2.1 Source code conversion scripts

In order to facilitate routine porting tasks we propose a toolkit-assisted approach. To determine the routine activities that porting specialists deal with we examined eight (8) scientific codes in ORNL production use from a wide spectrum of computational science (chemistry, biology, fusion, computer science, and climate). We focused on those applications with available baseline and ported source codes, relevant to ORNL computing systems (Cray X1E (vector machine), Cray XT3/4, IBM SP4 (PowerPC processors)). A PC Linux distribution of a scientific application served as a baseline code.

In general we can distinguish between two main code conversion categories, namely *automatic* and *manual*. The former refers to the set of conversions that to some extent may be automated, although user steering and input is still necessary. The simplest automatic conversions are *substitutions* that play a similar role to name refactoring (e.g., adding the prefix PXF for POSIX functions on Cray machines, or *identifier mangling conventions* for mixed-language codes). More advanced conversions concern *pattern mapping*. As an illustration, consider the different parameter passing conventions, e.g., the PGI Fortran compiler CALL FREE(PTR) and the IBM Fortran compiler CALL FREE(%VAL(PTR)), or time functions that may differ in semantics on various high-end machines. The other example relates to library incompatibilities such as a new version of the same library that has not been ported to a machine yet (e.g. FFTW 3.x $\mapsto$ 2.x), or highly vendor-optimized library counterparts.

Apart from mapping conversions there are cases where a given HPC system does not support certain features such as signals, threads, some system calls, sockets, or a synthetic file system (i.e., /proc). *Detec-*

tion conversions attempt to deal with such situations and inform the user about non-portabilities. In general, detections trigger manual code adaptations that usually require expert knowledge of the hardware (architecture), system software in terms of compiler switches, usage of relevant libraries or versions, and application algorithms. For instance, in order to utilize a streaming feature such as that provided by SSE or 3DNow, the algorithm must be implemented in an assembler code. Another example concerns code vectorization to fully exploit vector processors, manual loop unrolling, or performance optimization and tuning.

Based on the described source code conversions, we developed *conversion assistant modules* in Python and tested this approach on the CPMD application. We examined the number of modifications of the baseline source code that can be supported by our conversion modules in comparison to the total number of necessary modifications to successfully build and execute an application on a given high-end machine. As target platforms we chose HPC systems relevant to ORNL, i.e., IBM SP4, Cray X1E, and Cray XT3/4. We used a preprocessor to generate architecture-specific versions of application source codes. The target application chosen was CPMD since it supports, among others, architectures of interest to us. We assumed the baseline code is the CPMD PC Linux distribution. The detailed discussion of obtained results is presented in the PPAM paper [2]. In general, the results demonstrate that our methodology is promising and may even contribute much more to porting applications which are not as well prepared for this process as CPMD is (CPMD is portability-oriented, e.g. porting-sensitive routines or functions such as malloc() or open() are wrapped into proprietary functions). The outcome of this experiment shows that although manual conversions are much more cumbersome, require substantial effort and knowledge, and cannot be completely eliminated, we can identify many conversions that can be supported by a tool, and in this regard improve productivity of scientists involved in the porting process.

2.2.2 Late binding

In our approach, target-specific knowledge is encapsulated at the hardware, system, and application levels in structured (XML), declarative profiles, created by vendors, site administrators, developers and adjusted by users, and contain information about available compilers, compiler flags, library paths, environment settings, etc. These platform-specific values need to be applied eventually to the target machine. One step towards this direction is to develop the mechanism of dynamic concretization of platform-specific variables from profiles during the actual build or execution process.

In this approach, the original application build system needs to be generalized, i.e., modified in such a way that target platform specific data is replaced with references, called *vct references*, as presented in Figure 3. The build-related file content is concretized on demand, during file content reading. We call this mechanism *late binding* to emphasize its dynamic aspect.

To help address build configuration requirements resulting from heterogeneity issues, the common practice is to distribute application source codes with many configuration files for particular operating systems, compiler suites, hardware platforms, etc. Our approach aims to unify access to configuration information while preserving the current application build system that is capable of exploiting this information. This simplifies build configuration maintenance by allowing us to delegate it to a dedicated software service. In an ideal situation there will be one generalized build script instead of many predefined configuration sets, which, through the late binding mechanism are dynamically instantiated as appropriate to the target computing system requirements.

Our implementation of the late binding mechanism exploits FUSE (Filesystem in Userspace) that implements a fully functional userspace filesystem. Our prototype FUSE server – the VCT agent (implemented in Python 2.5.1 with the FUSE-Python module support) – intercepts invocations to filesystem operations that are performed on a source code directory, as shown in Figure 3. Generalized build files, stored at the target HPC system are concretized through profiles by the VCT agent after receiving the file read request from the OS kernel.

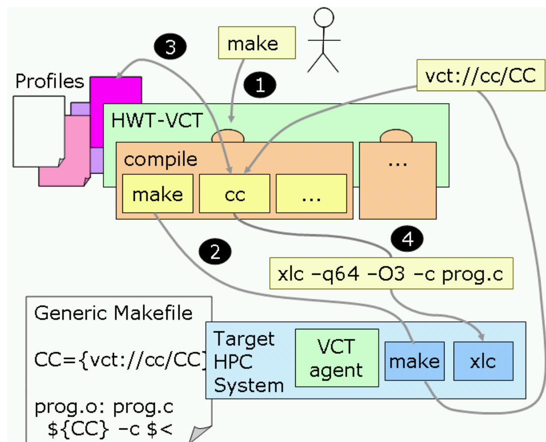

Figure 3: Late binding in the toolkit

In order to exploit FUSE, however, the FUSE module has to be added to the OS kernel. We note that FUSE support is included in the official Linux kernel from version 2.6.14 onwards.

To verify the feasibility of our approach we performed the build experiment with the CPMD application. First, we generalized the CPMD build system by creating a generalized (template) configuration file (GCF) with vct references resolvable by the HWT, e.g. CFLAGS={vct://cpmd app/CFLAGS} instead of a hard-coded option such as CFLAGS=-c -O2 -Wall. Then, we built the CPMD code with GCF and VCT agent, and specific values provided by the PC-GFORTRAN configuration file. The experiment was conducted in our local laboratory environment on PC workstations controlled by the Ubuntu 7.04 Linux (VCT agent) and Windows (HWT). The approximate impact of the VCT – HWT build model in comparison to the original version is reasonable and indicates that the generalized build system is only about 4% slower than the original despite a very simple implementation of the VCT agent.

We believe that an intercepting mechanism will be necessary to concretize target-specific values from relevant profiles and the proposed late binding mechanism takes a step towards achieving this goal.

2.3 HWT prototype in context of HWT architecture

At this project phase the HWT prototype consists of the graphical HWT core platform described in Section 2.2 and practical proof-of-concepts described in sections 2.2.1 and 2.2.2. In particular, the graphical HWT prototype constitutes the core platform that permeates all three HWT architecture layers shown in Figure 1. We believe that in its current version the prototype is functional and can considerably simplify routine build tasks. The core platform enables plugging actual modules responsible for porting conversions, preconditioning, compilation, installation, and execution on heterogeneous machines. Some of them are partially implemented as individual tools (e.g. Python porting assistant scripts) or their preliminary versions are provided by the platform (e.g. staging, launching). We are currently working on packaging the initial release of the HWT.

2.4 Encapsulating target-specific knowledge

As we described in the previous report the most important issue in design of the capability model is the *identification* of resource capabilities at different levels: hardware, system software, and application. We have working on a model to encapsulate target-specific knowledge at (1) the hardware level (machine's

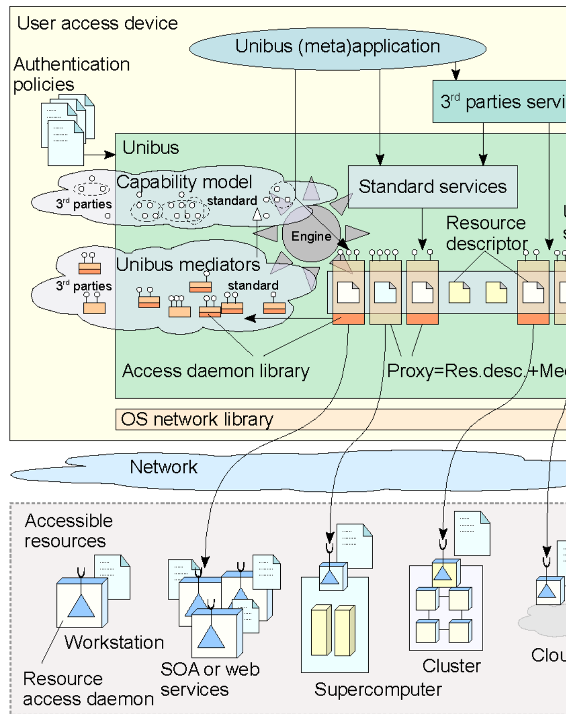

Figure 4: Unibus architecture

architecture type, type and number of processors, available RAM, etc); (2) the system software level (application launcher methods (ssh, job scheduling systems), libraries and their locations and versions, compilers, linkers, preprocessors); (3) the application level (tuning and optimization parameters, dependencies, deployment, environment settings). We note that the capability model should provide semantics on how the capabilities can be used. We have develop a tool based for creating and managing capabilities manifested in profiles. The profile syntax allows for manual modifications by end-users or by automated external tools. This tool has been tested on the Gamess-US application on the Jaguar XT5 machine at Oak Ridge National Laboratory.

2.5 Environment Conditioning with Unibus

The Unibus subsystem draws inspiration from the traditional VO (Virtual Organization) resource sharing model. Unlike in Grids, however, where resource virtualization and aggregation take place at the resource providers side and are performed by resource providers, the Unibus goal is to relieve them from that burden and shift it to software (Unibus) at the resource clients side. In this manner, Unibus benefits both resource providers and resource users, as the former expose their resources in an arbitrary way, and the latter can execute their applications on resources orchestrated by Unibus in accordance with their requirements.

The Unibus architecture is presented in Figure **??**. In Unibus, resources are exposed by resource providers through access points, typically represented by access daemons (e.g., sshd, ftpd, etc). Resources are described semantically by their OWLDL resource descriptors that contain resource-specific data related to authentication methods, available access points, installed system software (OS, libraries, compilers, etc), environment variables, etc.

Unibus provisions resources through a process known as conditioning that increases the resource spe-

cialization level. In particular, there are two classes of conditioning services: (1) soft conditioning, and (2) successive conditioning. Soft conditioning alters resource capabilities in terms of installed software, e.g., installing an MPI implementation on a resource or a gcc compiler, allows to execute MPI applications or compile C programs, respectively. Successive conditioning results in enhancement of resource capabilities in terms of access points, e.g., Globus Toolkit installation adds the Grid access point on the resource. Typically, successive conditioning will be supported by soft conditioning in order to deploy new access points on a resource. The typical Unibus usage scenario requires creation of a metaapplication that is executed on the local users machine. As the Unibus framework is implemented in Python, the most straightforward approach to implement the metaapplication is to write it as a Python script. We have implemented a prototype of this subsystem and have conducted preliminary experiments on several target platforms, including cluster systems, Amazon EC2 and Rackspace cloud systems.

3 Research Output

In the HWB project we have designed the VCT architecture to encompass portability issues in daily build tasks (HWT). We have also proposed and practically examined the toolkit-assisted approach to porting and the late binding mechanism to resolve generalized build systems. Finally, as part of the Unibus sub-project, we have developed semi-automated methods to condition target environments to be ready for application deployment. The results have been disseminated in the publications listed below.

1. Magda Slawinska, Jaroslaw Slawinski, Vaidy Sunderam, "A Practical SCVM-based Approach to Enhance Portability and Adaptability of HPC Application Build Systems", *Proc. 2012 International Conference on Computer Science (IMECS 2012)*, Hong Kong, pp. 257-262, March 2012.

2. Julien Bourgeois, Vaidy Sunderam, Jaroslaw Slawinski, Bogdan Cornea, "Extending Executability of Applications on Varied Target Platforms ", *Proc 13th IEEE International Conference on High Performance Computing and Communications (HPCC)* , Banff, Canada, September 2011.

3. Magda Slawinska, Jaroslaw Slawinski, Vaidy Sunderam, "Towards Cross-Platform Cloud Computing", *Proc. Workshop on Cloud Computing Projects and Initiatives*, 17th Euro-Par 2011 Conference, Bordeaux, France, August 2011.

4. Jaroslaw Slawinski, Magdalena Slawinska, Vaidy Sunderam, "Unibus-managed Execution of Scientific Applications on Aggregated Clouds", *Proc. 10th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing*, Melbourne, Australia, May 2010.

5. Magdalena Slawinska, Jaroslaw Slawinski, Vaidy Sunderam, "Aspects of Heterogeneity and Fault Tolerance in Cloud Computing", *Proc. Intl. Parallel and Distributed Processing Symposium (IPDPS-HCW) 2010*, Atlanta, GA, April 2010.

6. Magdalena Slawinska, Jaroslaw Slawinski, Vaidy Sunderam, "The Unibus approach to Aggregation of Heterogeneous Computing Infrastructures", *Proc. Intl. Conference on High Performance Computing Workshop on Utility and Grid Computing (HIPC-WUGC) 2009*, Cochin, India, December 2009.

7. Jaroslaw Slawinski, Magdalena Slawinska, Vaidy Sunderam, "Provisioning Software Applications on Diverse Resources", *Proc. Intl. Conference on High Performance Computing Workshop on Service Oriented Computing (HIPC-WSOC) 2009*, Cochin, India, December 2009.

8. Magdalena Slawinska, Jaroslaw Slawinski, Vaidy Sunderam, "Portable Builds of HPC Applications on Diverse Target Platforms", *Proc. Intl. Parallel and Distributed Processing Symposium (IPDPS-HCW) 2009*, Rome, Italy, pp. 1-8, May 2009.

9. M. Slawinska, J. Slawinski, V. Sunderam, “Enhancing Build-Portability for Scientific Applications Across Heterogeneous Platforms”, *Proc. Intl. Parallel and Distributed Processing Symposium IPDPS-HCW 2008*, Miami, FL, April 2008.

10. M. Slawinska, J. Slawinski, V. Sunderam, “Enhancing Productivity in High Performance through Systematic Conditioning”, *Parallel Processing and Applied Mathematics*, Gdansk, Poland, September 2007.

11. J. Slawinski, M. Slawinska, V. Sunderam, “Porting transformations for HPC applications”, *International Parallel and Distributed Processing Symposium*, Las Vegas, Nevada, September 2007.

12. M. Slawinska, J. Slawinski, D. Kurzyniec, V. Sunderam, “Enhancing portability of HPC applications across high-end computing platforms”, *Proc. Intl. Parallel and Distributed Processing Symposium HCW 2007*, Long Beach, CA, March 2007.

13. M. Slawinska, D. Kurzyniec, J. Slawinski, V. Sunderam, “Automated deployment support for parallel distributed computing”, *Proc. 15th Parallel, Distributed and Network based Processing*, Naples, Italy, February 2007.

14. D. Kurzyniec, M. Slawinska, J. Slawinski, V. Sunderam, “Unibus: A Contrarian Approach to Grid Computing”, *The Journal of Supercomputing: Special Issue on Grid Technology*, 42(1), 2007.

15. M. Slawinska, D. Kurzyniec, J. Slawinski, V. Sunderam, “Zero-Force MPI: Towards Tractable Toolkits for High Performance Computing”, Poster presentation, *Supercomputing 2006 (SC06)*, Tampa, FL, November 2006.

One postdoctoral fellow and one post-graduate research associate were supported by this award. In addition two graduate students were partially supported by this grant and worked on thesis projects related to this project.

4 Summary

This report has outlined the Harness workbench project accomplishments at Emory University. Additional information or copies of publications may be obtained from the principal investigator. U.S. Department of Energy support of our research efforts is greatly appreciated.