Predicting Execution Readiness of MPI Binaries with FEAM, a Framework for Efficient Application Migration

Karolina Sarnowska-Upton and Andrew Grimshaw
Department of Computer Science
University of Virginia
Charlottesville, VA

Abstract— **Today's scientific computing infrastructures provide scientists with easy access to a wide variety of computing resources. However, migrating applications to new computing sites can be tedious and time consuming. When optimal performance is not a concern, scientists can benefit by moving binaries instead of source code. Our work aims to make migration of MPI application binaries more efficient by automation. We present general methods that assess if binaries are a good match for execution at computing sites. We also present methods for increasing execution readiness by resolving missing shared libraries. Our work aims to free scientists from extensive manual preparation at new sites. To evaluate the effectiveness of our methods, we present an automated Linux-based implementation called FEAM, a Framework for Efficient Application Migration. We show that FEAM is more than 90% accurate at predicting execution readiness of MPI application binaries from the NAS Parallel and SPEC MPI2007 benchmark suites. In our evaluation, only half of the migrated binaries execute successfully at sites configured with a matching MPI implementation. We show that by automatically resolving shared libraries requirements, FEAM is able to increase the number of successful executions by a third.**

Keywords- execution prediction, environment configuration; automated methods; MPI; migration

I. INTRODUCTION

Today's scientific computing infrastructures, such as the Extreme Science and Discovery Environment (XSEDE) [1], provide scientists with easy access to a wide variety of computing resources. However, in order to use these diverse resources, scientists must ensure that their applications can execute at new computing sites. Today, the main users of these infrastructures are participants in "big science," i.e., projects with large budgets, extensive collaborations, large data generation, complex instruments and/or lengthy timescales. Although these communities are more likely to have experience with and support for application migration, they represent a minority of scientists. To enable more scientists to use various computational resources, ease-of-use needs to be increased. Our work aims to free scientists from extensive manual preparation at new computing sites.

Scientists can migrate application source code or binaries. When optimal performance is not a concern, scientists can benefit by moving binaries instead of source code. They can avoid long compile times or compiling community codes they did not author. They can gain quicker access to sites with more cores or sites experiencing shorter queuing delays. In this paper, we focus on the case when scientists migrate application binaries.

Before application binaries can run at new computing sites, the sites need to be configured with dependencies such as libraries, run-time environments, or other software required by the applications. Applications compiled with an MPI stack -- the combination of the MPI implementation, associated compilers, and interconnection network -- have an additional layer of dependencies. Without experience or support, scientists may need many hours to familiarize themselves with just one new environment, determine its configuration, and resolve dependencies. Considering this time requirement, preparing multiple sites manually does not scale well. These scientists would benefit from methods that make migration to otherwise easily accessible computing resources less tedious and less time consuming.

Our work aims to make migration of application binaries more efficient by automation. We describe general methods that assess if MPI application binaries are a good match for executing at computing sites. We also describe methods for increasing execution readiness by resolving missing shared libraries. We present and evaluate a Linux-based implementation of our methods called FEAM, a Framework for Efficient Application Migration. FEAM predicts execution readiness, resolves missing shared libraries, and automates site configuration.

Our results show that we can predict an application's execution readiness with more than 90% accuracy by only considering four basic characteristics of an application binary. We also show that ensuring the presence of shared libraries at computing sites increases the number of successful executions by more than 30% as compared to only ensuring a compatible MPI implementation is selected. Our results were gathered using MPI binaries from the NAS Parallel and MPI SPEC2007 benchmark suites. To the best of our knowledge, our implementation is the first to automatically predict application migration readiness and compose site-specific configurations. Existing technologies assume that developers or users can fully describe application characteristics as relevant for deployment. Other solutions assume that developers or users will create configuration procedures for new computing sites.

The remainder of this paper is organized as follows. We begin with a discussion of related work in the next section. We then present our proposed methods and their implementation. We describe our prediction model of execution readiness in Section III. We describe our resolution model for handling missing shared library dependencies in Section IV. We then explain how we implemented our methods in FEAM, our Framework for Efficient Application Migration, in Section V. We discuss our evaluation of FEAM's effectiveness using binaries from the NAS Parallel and MPI SPEC2007 benchmark suites in Section VI before concluding in Section VII.

II. Related Work

Various technologies exist to aid in the preparation of computing sites for the execution of applications. Commonly used HPC resource managers, like PBS [2], SGE[3], and SLURM [4], ensure applications are run on compute nodes with specified hardware characteristics like size of memory and number of nodes. They provide a mechanism with which users can deploy scripts to configure an environment further. Representation formats, such as SDD [5] and CDDLM [6], can be used to describe application information related to deployment. Frameworks, such as GLARE [7] and DistributedAnt [8], can be used to describe deployment instructions and perform them at new computing sites. There are also other technologies that utilize related methods while addressing different problems. Build tools, like Autoconf [9], gather information about execution environments to aid with resolving compile dependencies. Package managers provide tools for administrative management of software on computing systems [10,11,12]. Virtual machines mask the heterogeneity of execution environments while posing their own deployment limitations and incurring a performance overheard for many parallel applications [10]. H3wever, unlike our solution, existing technologies do not automatically identify and describe the application information related to deploying an application at a new site. It is assumed that application developers or users will describe application characteristics as relevant for deployment. Similarly, unlike our resolution techniques, existing technologies do not automatically compose site-specific configuration instructions. It is assumed that application developers or users will develop the deployment procedures for new computing sites. Existing technologies cannot be utilized as efficiently by scientists trying to run their application binaries in new environments as the automated methods we present in this paper.

III. Execution Prediction Model

To predict whether an application is ready for execution, we have created a model that attempts to answer four key questions: 1) Was an application compiled for a compatible ISA? 2) Is there a compatible MPI stack functioning at a new computing site? 3) Are an application's C library requirements met at a new computing site? 4) Are all the correct versions of the shared libraries an application was linked against available at a new computing site? The answers to these four questions compose the determining factors for our prediction model. Figure 1 visually presents these determinants and the information that needs to be gathered to make a prediction. All but one of the determinants is general and could be used to predict the execution readiness of any application. It is the question of the MPI stack compatibility that is specific to parallel applications that use MPI. In this section, we explain our model for determining if an application is ready for execution.

First we define terminology used throughout this paper. We use the term *target site* to refer to a new computing site where an application is migrated and execution readiness is to be predicted. Usually, there will also be at least one site where an application already runs successfully (the site from where the application is being migrated). We call this a *guaranteed execution environment* as successful execution is guaranteed to be able to occur in this environment. A guaranteed execution site can be but does not have to be the site where an application was compiled. To make a prediction, our methods only require access to an application binary and a target site. However, access to a guaranteed execution environment can provide information for making a better prediction.

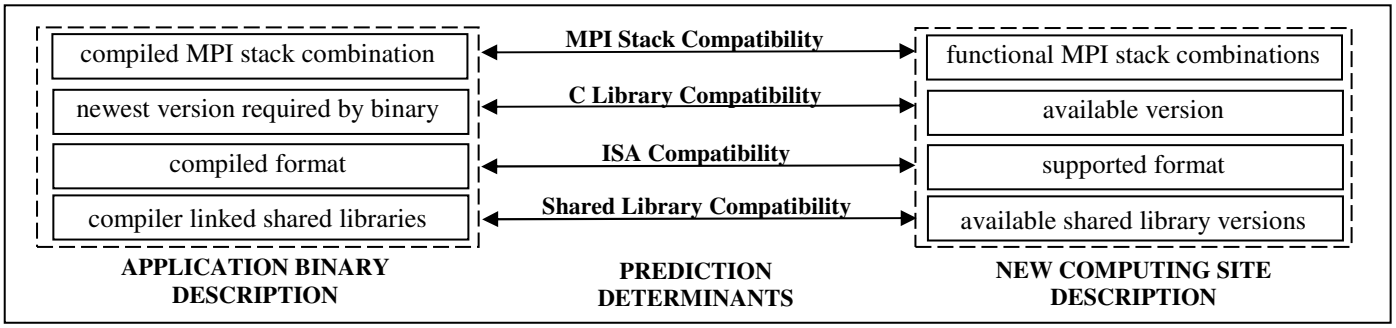

Figure 1. **Prediction Model Determinants**
Our prediction model determines an application binary's readiness to execute at a new computing site by answering four questions: 1) Does a compatible ISA exist? 2) Is there a compatible MPI stack functioning? 3) Are an application's C library requirements met? 4) Are all the correct versions of the shared libraries an application was linked against available?

A. ISA Compatibility

The most basic question we investigate is whether an application was compiled for an ISA that is compatible with a target site's architecture. Compatibility in this case refers to an application binary being compiled into a format that is executable at a target site from the hardware perspective. This investigation allows us to uncover incompatibilities between ISAs (i.e. ppc vs x86) as well as word-length (32 vs. 64-bit). The bitness information is also used when selecting between 32-bit and 64-bit shared libraries.

B. MPI Stack Compatibility

Since our work targets parallel applications that use MPI, we investigate if there is a compatible MPI stack - the combination of the MPI implementation, associated compilers, and interconnection network – functioning at a target site. When an application is not being recompiled at a target site or when an application has been dynamically linked, matching the MPI stack is critical because MPI is only an interface specification. The MPI standard specifies a library interface and not a library. Implementations of the standard have produced various libraries (i.e. Open MPI, MPICH, MVAPICH) that are not interchangeable because the MPI specification is not a link-level specification. As a result, each library has different dependencies. Similarly, matching the associated compiler (i.e. GNU, Intel, or PGI) and, if applicable, the interconnection network (i.e. Ethernet or Infiniband) is important in order to match other dynamically linked shared libraries.

We define a compatible MPI implementation as any implementation of the same type (i.e. an application compiled with Open MPI is only compatible with Open MPI and not MVAPICH or MPICH). In determining compatibility, the version of the MPI implementation is not considered as we have not found any guaranteed guidelines for backwards compatibility between different versions of the same MPI implementation. For example, we have found that an application compiled with Open MPI version 1.4 executes on a system using Open MPI version 1.3 in some instances but not others. However, we do consider version compatibly between shared library dependencies of MPI libraries. This compatibility is determined using the same methods as for any other shared library (described in Section III.D).

Our methods not only investigate the existence of a compatible MPI stack but also determine if the MPI stack combination is useable. We have found that even when an MPI stack combination (i.e. MVAPICH2 with the Intel compiler) is advertised by a target site, it is possible for the MPI stack combination to not be useable. Consequently, no programs are able to execute when that MPI stack is selected. One cause of these types of failures is misconfiguration by system administrators. For example, an MPI implementation or compiler version may be updated or a network may be reconfigured without testing all possible MPI stack combinations to ensure they still function. Our methods decide an MPI stack is useable if a basic MPI program is able to be executed when the MPI stack is selected. We attempt to compile the program natively but if that is not possible, we use basic MPI programs compiled at other sites to perform the test.

C. C Library Compatibility

C library compatibility is a major determinant in the ability of an application to execute at a new computing site. Most applications, as well as their shared library dependencies, are dynamically linked against a system's C library. Site administrators update their systems' C library versions at different times during the lifetime of a system resulting in various versions of the C library existing at target sites. Our prediction model examines whether an application's C library requirements are met at a target site.

An application binary has been compiled with a particular version of the C library. However, this is not necessarily the C library version that an application requires for execution. Rather, usually only the highest version of the C library that is actually used by an application must be available at a target site. We call this an application's *required C library version*. Our model considers a target site's C library version to be compatible if it is equal to or greater than an application's required C library version.

D. Shared Library Compatibility

The presence of shared libraries is the final determinant of our prediction model. Our methods ascertain if all shared libraries required by an application binary are available at a target site. Shared libraries compatibility is based on library naming and version conventions. Shared library names include major and minor release version numbers. The naming convention is of the format lib<name>.so.<major_version>.<minor_version>. Only shared libraries with the same major versions are guaranteed to have compatible APIs.

IV. RESOLUTION MODEL

The determinants of our prediction model are a result of the environment where an application was compiled. They cannot be changed without recompiling an application. However, a target site may be able to be altered to match the requirements of each determinant. This would be an involved process for the first three determinants. Matching an ISA would require emulation. Matching an MPI stack would require administrative privileges on a system to setup an MPI implementation. Matching a C library version requirement would involve installing the specific required version. However, matching a required shared library can often be accomplished by simply making a copy of the missing library available at runtime. This is the aim of our resolution model.

Our resolution methods copy shared libraries from an application's guaranteed execution environment. (Licensing issues are out of scope of our work.) If any required shared libraries are missing at a target site, our methods determine if the library copies can be installed to resolve the issue. To determine if a shared library copy can be used, we determine if a given library will execute at a target site. We apply the same analysis to these missing shared libraries as to any application binary. Our prediction methods are applied recursively to determine if a shared library copy is able to execute at a target site. This may include recursively resolving any missing shared libraries that the library copy requires. If a library copy is determined to be useable at a target site, our resolution methods make the library accessible at runtime by setting the appropriate environment variables.

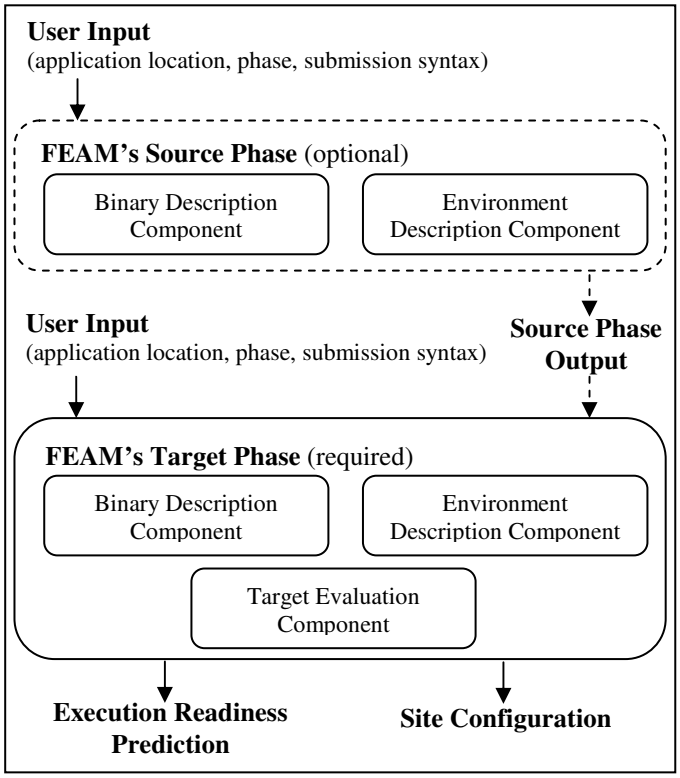

Figure 2. **The Phases and Components of FEAM**
FEAM, our Framework for Efficient Application Migration, consists of two phases and three components. The source phase is optionally run a guaranteed execution environment while the target phase must always be run at a target site. Running both phases enables our resolution methods and additional compatibility tests to be applied.

V. IMPLEMENTATION

We present the implementation of FEAM, a Framework for Efficient Application Migration. FEAM realizes our prediction and resolution models in an automated fashion. FEAM is composed of three components and two phases as illustrated in Figure 2. The *Binary Description Component* gathers information about an application binary and its dependencies. The *Environment Discovery Component* gathers information about a computing environment. The output from these two components is used by the *Target Evaluation Component* to determine whether execution can occur at a target site without recompilation. FEAM can be run in one or two phases. The *target phase*, which is required, is run at each target site. The *source phase*, which is optional, is run only once per application binary at a guaranteed execution environment. However, if the source phase does not occur, our resolution methods and the extended compatibility tests cannot be applied.

- ISA and file format of binary
- Library name and version, if applicable
- Required shared libraries, with copies and descriptions if applicable
- C library version requirements
- MPI stack, operating system, and C library version used to build binary

Figure 3. **Information Gathered by the BDC**
The Binary Description Component (BDC) of FEAM gathers information about MPI application binaries.

Running both phases of FEAM provides the additional benefit of not requiring the application binary to be present at a target site.

Before running FEAM, a user needs to specify (via a configuration file) a serial and parallel submission script for the site. The submission format is the only information about a new site our methods require the user to determine. The user also specifies what FEAM phase is to be run and the location of the application binary if applicable. The output from a source phase is bundled for the user and must be copied to each target site if it is to be used in a target phase. The output from a target phase is a prediction of execution readiness along with a matched configuration of a target site if execution is predicted to be possible.

In this section, we describe the techniques that each of the FEAM components applies. Our methods are implemented using various standard Unix-like operating system utilities. This limits the current version of FEAM to working on Unix-like operating system but in general our methods could be implemented for any system. The information needed by our prediction and resolution models is gathered in multiple ways by FEAM in case some tools are not present or functioning at a particular target site. While our techniques are relatively simple, their composition efficiently and automatically provides a determination of whether a migrated application is ready to execute without recompilation along with the matching site configuration details.

A. *Binary Description*

The Binary Description Component (BDC) of FEAM gathers information about an application binary and its dependencies. It expects as input the location of the binary. To be able to apply all of our methods, the BDC needs to execute in a guaranteed execution environment and at a target site. However, only execution at a target site is required to enable a basic prediction. The information collected by the BDC is listed in Figure 3. In this subsection, we explain how we gather this information.

TABLE I. IDENTIFYING LIBRARIES OF MPI IMPLEMENTATIONS

MPI Implementation	Library Dependencies
MVAPICH2	libmpich/libmpichf90, libibverbs, libibumad
Open MPI	libnsl, libutil
MPICH2	libmpich/libmpichf90 (and not other identifiers)

Most of the information about a binary can be extracted with the GNU Binary Utility [14] `objdump`. The BDT calls the GNU Binary Utility `objdump` with the "-p" flag to view information that is specific to the file format of a binary. The format description specifies the file format (i.e. ELF) of a binary as well as for what ISA and for how may bits an application was compiled. We create a list of any shared libraries an application binary has been linked against from the "NEEDED" components under the "Dynamic Section" of the description. We calculate an application's C library version requirement by determining the newest version listed under the "Version Definitions" and "Version References" sections of the description. If a binary being analyzed is actually a shared library, we additionally capture the library's official shared object name from the "Dynamic Section" and extract from it the embedded version information.

The Unix utility `ldd`, when called with the `-v` command-line option, can also be used to view the shared library dependencies of dynamically linked binaries along with their C library version requirements. However, there are cases for which this tool does not recognize an application as being dynamically linked and, thus, cannot be relied on to always provide this information.

If an application binary is an ELF file (the standard binary format for Unix and Unix-like system on x86), the BDT calls the GNU Binary Utility `readelf` with the "-p .comment" flag to display the contents of the comments section. The optional comments section may contain compiler and linker specific version control information. This information is used to indicate under what OS and with what C library version an application binary was created.

To identify the MPI implementation used to compile an application, we examine the list of shared libraries that an application was linked against. Our identification scheme takes advantage of the fact that MPI is not a link-level specification. Rather, the MPI standard specifies a library interface and consequently implementations of the standard have different link-level dependencies. We have created an identification scheme for the three dominant open source MPI implementations (Open MPI [15], MPICH2 [16], and MVAPICH2 [17]). The shared libraries that we use as MPI implementation identifiers are listed in Table I.

- ISA format
- Operating system
- C library version
- Available or currently loaded MPI stacks
- Missing shared libraries

Figure 4. **Information Gathered by the EDC**
The Environment Description Component (EDC) of FEAM gathers information about a computing site.

When running at a guaranteed execution site, the BDC also gathers a description and copy of all of the shared libraries an application has been linked against. Each library goes through the same description process as an application binary. To locate the shared libraries for copying, we use the `ldd` utility. This utility lists the shared libraries an application has been linked against along with their locations in the local file system. We copy each shared library except for the C library. If the `ldd` utility is unable to provide location information, then we perform a search for each shared library. We apply three search methods to locate each library from the list gathered using `objdump`. If available, the `locate` utility is used to reveal the locations of files with matching names. The `find` utility is used to search common library locations as well as locations set in the LD_LIBRARY_PATH environment variable. If a locally compiled "hello world" program is available, the `ldd` utility is used to reveal the locations of commonly linked against shared libraries.

B. Environment Discovery

The Environment Discovery Component (EDC) gathers information about a computing environment after the BDC has created a description of an application. The EDC can gather information about a target site or a guaranteed execution environment. The type of environment must be identified by the user as it affects the discovery process. The information gathered by the EDC is listed in Figure 4. The EDC determines what operating system and hardware architecture is present at a computing site. The EDC also determines what version of the C library and what types of MPI stacks are configured on a system. The EDC may also compile and run "Hello World" programs to test the environment. In this subsection, we explain what techniques are used to discover this information.

The EDC calls the Linux utility `uname` with the "-p" flag to determine the system's ISA format details. We also examine standard files with system information under `/proc` and `/etc` to determine which Linux distribution is running on the system. We consult the `/proc/version` file to determine the OS type and version information. We confirm this information by examining files under `/etc/*release`. The distribution information is gathered only to provide the user with more information about a system.

The EDC determines the version of a system's C library by parsing the general library information that is output when C library binary is executed. The location of the library is found by using the same methods used by the BDC to search for shared library locations. If the C library binary cannot be run on the command line, the EDC attempts to determine the version using the C library API.

The type of information we gather about the local MPI stack depends on what type of site where the EDC is being run. At a target site, we search for available MPI stacks. In a guaranteed execution environment, we confirm if the MPI stack currently selected to run an application matches the stack combination that was discovered by the BDC. We also generate MPI "hello world" programs for later testing.

To determine what MPI stacks are available in a computing environment, user-environment management tools are consulted. These tools support the discovery of packages and help manage the shell environment. We specifically search for the presence of `Environment Modules` [18] or `SoftEnv` [19] configuration files to assess if these user-environment management tools are present. If one of these tools is located, then we use their search mechanisms to locate MPI implementations and compiler combinations. If no user-environment management tools are found, then we use the same search methods as used by the BDC to locate shared libraries. We search for libraries that are distributed with each MPI implementation such as *libmpi* or *libmpich*. We also search for commonly used wrappers for compiling MPI programs (i.e. mpicc, mpif90). Often information about compilers associated with a particular MPI implementation is revealed by the naming scheme of a location path as well as by compiler wrapper version information. For example, the path `/opt/openmpi-1.4.3-intel/lib/libmpi.so` reveals that Open MPI is available for the Intel compiler. Running `/opt/openmpi-1.4.3-intel/bin/mpicc –v` reveals that version 11.1 of the Intel compiler is used for compilation.

To determine what MPI stacks are currently accessible in a computing environment, we examine the environment settings. If user-environment management tools are present, we use the corresponding mechanisms to reveal these setting. For example, if `Environment Modules` are present, we use the `module list` command to get a listing of what MPI implementation and compiler the shell is configured to access.

Alternately, we can search for MPI implementations that are accessible via the `PATH` and `LD_LIBRARY_PATH` environment variables. We can also determine what MPI implementation and compiler is referred to by commonly used wrappers for compiling MPI programs.

We use the `ldd` tool to identify if any shared libraries that an application was linked against are missing at a target site. If the `ldd` tool is unable to produce this information, we search for the libraries using the same methods as used to locate shared libraries by the BDC.

C. Target Evaluation

The Target Evaluation Component (TEC) uses the information gathered by the BDC and EDC to determine whether execution can occur at a target site without recompilation. The TEC determines the outcomes of our prediction and resolution models by using the information gathered by the BDC and EDC. The TEC executes at a target site.

The information gathered about an application's ISA format, C library version, and MPI stack requirements is matched with the information about what is available at a target site to determine compatibility as outlined by our prediction model. If the ISA and C library version determinants are found to be compatible, we proceed to evaluating the MPI stack and shared library determinants. If at any point we determine that execution cannot occur, the reasons are detailed to the user via an output file.

For each compatible MPI stack that is detected, we run a series of "hello world" programs to test if the stack is functioning as described by our prediction model. We compile "hello world" programs at a target site and test for successful execution to confirm the functionality of the MPI stack. If "hello world" programs from a guaranteed execution environment are available, we run them to confirm the compatibility between the selected MPI stack and the MPI stack used to compile an application. Running these tests assumes knowing the execution command for a particular MPI stack. Our methods by default will use the *mpiexec* command for execution while allowing the user to specify otherwise (per MPI type if necessary) via a configuration file.

Finally, if a compatible MPI stack has been determined to be functioning, we attempt to resolve any missing shared libraries. Resolution can proceed if a Source Phase has occurred. In phase I, our framework runs the BDC and EDC at a guaranteed execution site to gather copies and descriptions of required shared libraries. With this information, our resolution model can be applied. For any missing shared library, we recursively apply our prediction model to determine if the library copy can be used. If a library copy is determined to be useable at a target site, we make the library accessible at runtime by adding its location to the `PATH` environment variable. If our resolution techniques succeed for all missing libraries or if there were no missing libraries, then we predict that a target site is ready for application execution. We provide a description of the matching configuration details to the user along with a script that will set them up automatically on execution.

VI. Evaluation

In this section, we present an evaluation of our prediction and resolution methods as implemented in FEAM. We predict the execution readiness of MPI application binaries from two benchmark suites across five computing sites. To determine the effectiveness of our prediction model, we compare our prediction of execution readiness with whether an application was actually able to execute. We evaluate the impact of our resolution model by comparing how many binaries are able to execute when our techniques are and are not applied.

A. Test Set Description

We created MPI application binaries for the evaluation tests from two benchmark suites: the NAS Parallel Benchmarks and SPEC MPI2007. The NAS Parallel Benchmarks (NPB) suite [17] consists of applications derived from computational fluid dynamics. We used version 2.4 of the MPI reference implementation of the NPB suite. SPEC MPI2007 [18] is a benchmark suite developed from native MPI-parallel end-user applications. To create the MPI application binaries for our tests, we compiled the benchmarks with multiple MPI stacks at each target site as indicated in Table II. Some benchmarks would not compile with certain MPI stacks combinations while other binaries would not run at the site where they were compiled. This is why our final test set, with 110 NPB binaries and 147 SPEC MPI2007 binaries, is composed of a subset of the benchmarks suites. From the NPB suite, our test set consisted of four kernels (integer sort, embarrassingly parallel, conjugate gradient, and multi-grid on a sequence of meshes) as well as three pseudo applications (block tri-diagonal solver, scalar penta-diagonal solver, and lower-upper Gauss-Seidel solver). From the SPEC MPI2007 benchmark suite, our test set consisted of a quantum chromodynamics code (104.milc), two computation fluid dynamics codes (107.leslie3d and 115.fds4), a parallel ray tracing code (122.tachyon), a molecular dynamics simulation code (126.lammps), a weather prediction code (127.GAPgeofem), and a 3D Eulerian hydrodynamics code (129.tera_tf).

Five computing environments were chosen as target sites for the evaluation tests. The target sites were chosen

such that a diverse test set was created in terms of the operating system, hardware architecture, network interconnect, and MPI implementation. Three of the chosen test sites are state-of-the-art high performance computing systems available to researchers via the national XSEDE infrastructure [22]. They represent the main types of systems architectures available at the national supercomputing centers: symmetric multiprocessing (SMP), massively parallel processing (MPP), and hybrid CPU/GPU systems. The other two test sites are mid-size university clusters. One is part of the FutureGrid Project [23] test-bed while the other is a University of Virginia resource [24]. Table II details the characteristics of each of these computing sites as well as the MPI stack combinations used to create application binaries. The test sites run different versions of three Linux-based operating systems and the C library. The sites support three open source MPI implementations. Open MPI is available at five sites, MVAPICH2 is available at four sites, and MPICH2 is available at two sites. Each MPI implementation is associated with GNU, Intel, and/or PGI compilers for C and/or Fortran.

B. Methodology

To evaluate the effectiveness of our prediction methods, we compared predictions with actual execution results. These results are presented in Table III. We migrated each MPI application binary to all target sites where the binary had not been compiled. Our methods were 100% accurate at assessing whether a matching MPI implementation was available for all target set. However, we only report prediction results for sites with matching MPI implementations. Only at such sites is there potential for successful execution. For example, binaries compiled for MVAPICH2 on Ranger were tested only on Forge, India, and Fir while binaries compiled for MPICH2 on India were tested only on Fir. If results for all sites were reported, our prediction accuracy would be much higher. However, it is more interesting to focus on the accuracy of our model at sites where there is a potential for successful execution.

We formed predictions using information gathered at guaranteed execution environments and target sites. The results in Table III distinguish how predictions were made. Our *basic prediction* results were formed by only running FEAM's required target phase at a target site. Our *extended prediction* results were formed by also running FEAM's optional Source Phase at guaranteed execution environments. In this way, the results differentiate the effectiveness of our methods for instances when users do not have access to or do not want to access guaranteed execution sites. This situation in particular applies to community codes distributed as binaries.

TABLE II. TARGET SITE CHARACTERISTICS

Computing Site (Type - CPUs)	Operating System	C Library & Compiler Versions	Utilized MPI Stacks: MPI Types & Versions (Compilers i:Intel, g:GNU, p:PGI)
XSEDE Ranger, Texas Advanced Computing Center (MPP – 62,976)	CentOS 4.9	LibC v2.3.4, GNU CC v3.4.6, Intel v10.1	Open MPI v1.3 (i/g/p) MVAPICH2 v1.2 (i/g/p)
XSEDE Forge, National Center for Supercomputing Applications (Hybrid - 576)	Red Hat Enterprise Linux Server 6.1	LibC v2.12 GNU CC v4.4.5 Intel v12	Open MPI v1.4 (g/i) MVAPICH2 v1.7rcl (i)
XSEDE Blacklight, Pittsburg Supercomputing Center (SMP – 4,096)	SUSE Linux Enterprise Server 11	LibC v2.11.1, GNU CC v4.4.3, Intel v11.1	Open MPI v1.4 (i/g)
FutureGrid India, Indiana University (Cluster - 920)	Red Hat Enterprise Linux Server 5.6	LibC v2.5, GNU CC v4.1.2, Intel v11.1	Open MPI v1.4 (i/g) MVAPICH2 v1.7a2 (i/g) MPICH2 v1.4 (i/g)
ITS Fir, University of Virginia (Cluster - 1,496)	CentOS 5.6	LibC v2.5, GNU CC v4.1.2, Intel v12	Open MPI v1.4 (i/g/p) MVAPICH2 v1.7a (i/g/p) MPICH2 v1.3 (i/g/p)

TABLE III. ACCURACY OF PREDICTION MODEL

Basic Prediction		Extended Prediction	
NAS	SPEC	NAS	SPEC
94%	92%	99%	93%

TABLE IV. IMPACT OF RESOLUTION MODEL

Actual Execution Successes				Increase in Successful Executions due to Resolution	
Before Resolution		After Resolution			
NAS	SPEC	NAS	SPEC	NAS	SPEC
58%	47%	78%	66%	33%	39%

To evaluate the impact of our resolution methods, we calculated the percentage of successful executions occurring before and after applying our resolution methods. These results are presented in Table IV. The table also presents the overall increase in successful executions. This percentage was calculated as the increase in successful executions after applying our methods divided by the number of successful executions before applying our methods. As with our prediction results, we only report resolution results for sites with matching MPI implementations where there is a potential for successful execution.

C. Results Analysis

Our predictions were over 90% accurate for both basic and extended prediction schemes. The extended prediction created from running FEAM at guaranteed execution environments along with target sites increased

prediction accuracy due to additional compatibility tests. In particular, by testing the execution of MPI "hello world" programs compiled at guaranteed execution environments, we were able to detect floating point errors and application binary interface (ABI) incompatibilities in shared libraries. Our model was unable to predict failures due to system errors such as failed MPI daemon spawning or time-outs due to communication errors. If execution was not successful after five execution attempts, we classified the actual execution result as failed. We spaced retries in time to avoid errors caused by an overload in the system.

Around half of the MPI application binaries were able to execute at target sites after migration (58% of the NAS binaries and 47% of the SPEC binaries). These percentages do not include the large number of target sites where binaries could not execute due to lack of a matching MPI implementation. Our results illustrate that more than just the MPI implementation determines whether a binary can execute at a target site. Of the failing jobs, more than half were missing shared libraries. Scientists compiling their own or community MPI applications at sites where MPI implementations have not been installed with static libraries do not have the option to prepare statically linked binaries for migration. The remaining jobs failed due to C library version requirements, floating point exceptions, and system errors.

Our resolution techniques automatically enabled execution for about half of the binaries that would have otherwise failed due to missing shared libraries. This resulted in enabling around a third more successful executions overall (33% for the NAS binaries and 39% for the SPEC binaries). The other half of the missing library failures could not be resolved mainly due to incompatibility issues. The shared libraries copies we had gathered at guaranteed execution sites for use in resolution required incompatible C library versions and used incompatible application binary interfaces. The remaining resolution attempts failed due to system errors.

Since running on compute nodes does use allocation hours (if any usage accounting is in place at a site), we measured how many CPU hours were used by running FEAM. We found that both FEAM's source and target phases always took less than five minutes to complete. This makes FEAM ideal for submission via a debug queue at sites. We also measured that a bundle of shared library copies composed by FEAM's source phase averaged 45M in size. This bundle consisted of all the shared libraries required by all of our test binaries at a site. Thus, the size of a bundle for just one MPI application binary would be much smaller.

VII. CONCLUSIONS

In this paper we presented methods that provide an automated and efficient means of determining where MPI applications can run without recompilation. We described methods that can be used to assess whether MPI application binaries can execute at new computing sites. We also described methods that increase the likelihood an application binary will execute at a new computing site by gathering and resolving shared library requirements. We presented a Linux-based implementation of these methods called FEAM, a Framework for Efficient Application Migration. We evaluated FEAM across five computing sites. A test set of MPI binaries was created by compiling applications from NAS Parallel and MPI SPEC2007 benchmark suites with three open-source MPI implementations and three compilers. Overall, our prediction results were more than 90% accurate. We found that choosing an execution site only by matching the MPI implementation resulted in half of the binaries failing due to other errors. More than half of these failures were caused by missing shared libraries. FEAM enabled about a third more successful executions at our test sites by automatically resolving missing shared libraries. FEAM was able run our prediction and resolution techniques in less than five minutes, making it a good candidate for running via a debug queue when available.

FEAM executes with minimal input from the user, runs quickly, and requires little space. It relieves the user of manually parsing various environment configurations to find out if a site is a good match for execution. For scientists who do not have much experience, time, or support to explore new computing sites or recompile their MPI applications, FEAM provides an efficient automated solution for quickly assessing many new computing sites.

The general direction of our future work will be to develop more methods for efficiently migrating MPI applications to new environments. This will include migrating MPI application binaries as well as MPI application source code. We are also interested in quantifying the amount of user effort required to perform migration tasks so that we can more concretely compute the efficiency gains of using our methods.

ACKNOWLEDGMENT

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575. This work also used resources supported in part by the National Science Foundation under Grant No. 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Testbed."

REFERENCES

[1] XSEDE: Extreme Science and Engineering Discovery Environment. https://www.xsede.org/.

[2] L. Bayucan, *et al*, "Portable Batch System External Reference Specification", MRJ Technology Solutions, May 1999.

[3] W. Gentzsch, "Sun Grid Engine: towards creating a compute power grid", Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid, 2001, pp.35-36.

[4] SLURM: Simple Linux Utility for Resource Management, http://www.llnl.gov/linux/slurm/slurm_design.pdf

[5] J. McCarthy and B. Miller. Solution Deployment Descriptor (SDD), Part 1: An emerging standard for deployment artifacts. IBM DeveloperWorks. 2008.

[6] A. Dantas, et al. "Using web services for configuration and deployment according to the CDDLM standard," International Conference on Web Services, 2006, pp. 951-954.

[7] M. Siddiqui, et al. "GLARE: A grid activity registration, deployment and provisioning framework," Proceedings of the ACM/IEEE SC 2005 Conference, 2005, pp. 52 - 64.

[8] W. Goscinski and D. Abramson. "Distributed Ant: a system to support application deployment in the grid," Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing, 2004, pp. 436-443.

[9] Autoconf - GNU Project. http://www.gnu.org/software/autoconf/.

[10] E.C. Bailey. Maximum RPM: Taking the Red Hat Package Manager to the Limit. Red Hat, Inc., 2000.

[11] Yum: Yellow Dog Update Modifier. http://yum.baseurl.org/.

[12] Smart Package Manager. http://labix.org/smart/.

[13] C. Xu, Y. Bai, and C. Luo. "Performance envalutation of parallel programming in virtual machine environment," Sixth IFIP International Conference on Network and Parallel Computing, 2009, pp. 140-147.

[14] GNU Binutils. http://www.gnu.org/software/binutils/.

[15] OpenMPI: Open Source High Performance Computing. http://www.open-mpi.org/.

[16] MPICH2: High Performance and Highly Portable MPI. http://www.mcs.anl.gov/research/projects/mpich2/.

[17] MVAPICH2: MPI-2 over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/overview/mvapich2/.

[18] Modules – Software Environment Management. http://modules.sourceforge .net/.

[19] Msys - The MCS Systems Administration Toolkit. http://www.mcs.anl.gov/hs/software /systems/msys/.

[20] R.F. Van der Wijngaart. *NAS Parallel Benchmarks Version 2.4*. 2002.

[21] M.S. Müller, et al. "SPEC MPI2007 - an application benchmark suite for parallel systems using MPI." *Concurrency and Computation: Practice and Experience* Vol. 22, Issue 2 (2010): 191-205.

[22] XSEDE Resources Overview. https://www.xsede.org/resources/overview

[23] FutureGrid: A Distributed Testbed for Clouds, Grids, and HPC. https://portal.futuregrid.org/.

[24] University of Virginia Alliance for Computation Science and Engineering: Resources. http://www.uvacse.virginia.edu/resources/.