Community Use of XALT in its First Year in Production

Reuben Budiardja
University of Tennessee Knoxville
reubendb@utk.edu

Mark Fahey
Argonne National Laboratory
mfahey@anl.gov

Robert McLay
Texas Advanced Computing Center
mclay@tacc.utexas.edu

Prasad Maddumage Don
Florida State University
mhemantha@fsu.edu

Bilel Hadri
King Abdullah University of Science and Technology
bilel.hadri@kaust.edu.sa

Doug James
Texas Advanced Computing Center
djames@tacc.utexas.edu

ABSTRACT

XALT collects accurate, detailed, and continuous job-level and link-time data and stores that data in a database; all the data collection is transparent to the users. The data stored can be mined to generate a picture of the compilers, libraries, and other software that users need to run their jobs successfully, highlighting the products that researchers use. We showcase how data collected by XALT can be easily mined into a digestible format by presenting data from four separate HPC centers. XALT is already used by many HPC centers around the world due to its usefulness and complementariness to existing logs and databases. Centers with XALT have a much better understanding of library and executable usage and patterns. We also present new functionality in XALT – namely the ability to anonymize data and early work in providing seamless access to provenance data.

Keywords

XALT user environment, library tracking, job analytics

1. INTRODUCTION

XALT collects accurate, detailed, and continuous job-level and link-time data and stores that data in a database; all the data collection is transparent to the users [1]. The data stored can be mined to generate a picture of the compilers, libraries, and other software that users need to run their jobs successfully, highlighting the products that researchers do and do not use.

In this work, we showcase how data collected by XALT can be presented into an easily digestible format. XALT is already used by several HPC centers around the world due to its usefulness and complementariness to existing logs and databases. With XALT, centers can have a much better understanding of how users use libraries and executables. In the examples shown, some are the results of queries done directly in the XALT MySQL database and the results dumped into a spreadsheet to make a pretty graph or pie chart. There are also examples using more sophisticated python scripts to do queries. We also include a section on how results can be anonymized in order to share data with others and early work in providing an easy seamless way to access provenance data.

Section 2 describes XALT and how it works including an overview of the database. Section 3 shows reports on compiler and library usage to demonstrate what can be done with XALT data, and describes the queries and scripts used to produce them. Section 4 also shows some more reports provided by community users of XALT and lets us begin to understand usage more broadly. Section 5 contains a brief description and example of the newest features in XALT that can be used to anonymize data and access provenance data. Section 6 has concluding remarks.

HUST2015 November 15-20 2015, Austin, TX USA

ACM ISBN 978-1-4503-4000-7/15/11…$15.00

DOI: http://dx.doi.org/10.1145/2834996.2835000

2. XALT OVERVIEW

XALT is designed to track linkage and execution information for applications that are compiled and executed on any Linux cluster, workstation, or high-end supercomputer. As such, XALT can be used to generate metrics intended to improve training, documentation, management and outreach programs.

2.1 Approach

The approach is based on wrappers that intercept both the GNU linker (ld) at link time and the code launcher (e.g. mpirun, aprun, srun or ibrun) at runtime. XALT targets common architectures, though subsequent releases will expand the list of supported systems as time permits. Wrapping the linker and the code launcher is a clean and efficient way to intercept information automatically and transparently, since many computational experiments will require both at some point in the workflow.

XALT tracks static and shared libraries specified at link time. In a future release XALT will also detect and store function calls that need to be resolved by external libraries.

Our wrapper for the linker (`ld`) intercepts the user link line and parses the command line, storing the results in a JSON[1] file within the user's XALT directory. At the same time an ELF section header is inserted into the user's code, which is a marker that will be used to record any subsequent use of this specific executable. As a secondary measure, we intercept the code launcher (aprun, mpirun, ibrun) at execution time via a wrapper. The script extracts all environment variables available at execution time, including job-specific environment variables from the batch system, such as job id (for examples `PBS_JOBID` in the case of PBS). It also detects dynamic libraries loaded during the runtime.

XALT can transmit the data in one of three ways to a database: (1) write to intermediate JSON files and parse later, (2) store directly in the database, or (3) write to syslog and parse later. The JSON files (i.e. the files created each time a code is compiled and executed) are then later stored in a database by running a script. Very similarly, the data can be written to syslog and later parsed and stored into the database. The motive for having three different methods is to let each site choose for themselves what works best in their environment based on their preferred file system and database accesses and security policy. For example, the motive behind storing the information in the JSON file is to limit the number of times the database is accessed. Accessing the database in real time may be of concern to centers, since thousands of users could be accessing the database at the same time. The process of reading the JSON files and storing the results in the database can be automated and scheduled during non-peak hours so that there is no adverse impact on the user experience. On the other hand, the direct to database method avoids placing potentially hundreds of files on the filesystem and then later parsed. The recommended best practice is using the syslog method, which utilizes standard logging techniques, avoids too many filesystem accesses, and limits the number of database accesses. This part of the XALT code base was written to be extensible and new methods could easily be added.

XALT is implemented so that:

- *Avoid impacting the user experience*: A primary design goal was that no matter what the XALT wrapper does (successfully or otherwise), it must not change the way users compile, link, and run their applications. This requirement is the overriding principle underlying the design of the XALT infrastructure.

- *Must work seamlessly on any cluster, workstation or high-end computer*: Computational research takes place on a wide spectrum of systems. XALT is a practical and usable solution on any Linux-based system.

- *Must support both static and dynamic libraries*: XALT tracks libraries during the linking process, and detects both static and dynamic libraries. In addition, XALT detects dynamic libraries that are loaded during the execution phase. With this data, one can determine if an executable uses different versions of dynamic libraries over its lifetime.

- *Lightweight solution*: XALT is a lightweight solution with essentially no overhead at compilation and runtime.

- *Support intercepting more than one linker and/or job launcher*: Some clusters have several MPI installations (MPICH2, MVAPICH, etc.) with multiple versions. Each such installation is likely to have its own launcher (e.g. mpirun). To be viable solution on any cluster, we must be able to intercept each code launcher.

- *Not everything on the original link line should be captured;* Often libraries are specified on the original link line that are not actually linked into the application. XALT excludes anything that is not actually linked to the executable.

- *Track library versions if possible*: XALT may or may not be able to track version information; much depends on how libraries are installed and managed at a given site. For example, some centers use modulefiles to provide environment variables with paths to libraries and applications with version numbers embedded in the paths.

2.2 Database Design

The XALT database has several tables; the basic layout is shown in Figure 1. Since there are several many-to-many relationships, we have designed the database with "join" tables that reduce the need to store redundant information. We have four core tables: `xalt_env_name`, `xalt_run`, `xalt_object`, and `xalt_link`. When a linker builds a program or shared library an entry is added to the `xalt_link` table. This table contains information about the program such as the builder, a time stamp, and what binaries linked to it. There will be multiple object files and libraries that constitute the built program. The `xalt_object` table contains information about the object path, system on which the object file is present, a unique hash id and the library type associated with the object. There is only one entry in the `xalt_object` for each object path that has the same sha1sum[2]. As there would be a number of object files and libraries linked to a program we will have a list of `object_id` associated with `link_id`s which would be stored in `join_link_object`. Data redundancy is avoided by the use of join tables, as we can have the same object files or libraries getting linked to different programs. The join table named `join_link_object` connects the `xalt_link` entry to the object files that are associated with a program. The `xalt_object` table also includes a module name field for the name of the modulefile that is associated with the object (if it exists).

When a program is executed via a launcher, we create a single entry in the `xalt_run` table. The `xalt_run` entry includes job statistics and execution information such as the job ID, the system host, account, start and end time et al. Additionally, shared libraries that belong to that executable (determined from `ldd`) are placed into the `xalt_object` table. The information stored is very similar to what we have discussed above. The entries made in `xalt_object` and `xalt_run` are related via the `join_run_object` join table. We also record the environment variables in the runtime environment in the `xalt_env_name` table and use the `join_run_env` table to connect the values of the variable to the name and job. The field `env_name` holds the variable

[1]JSON is an open standard data interchange format, see http://json.org

[2]linux.about.com/library/cmd/blcmdl1_sha1sum.htm

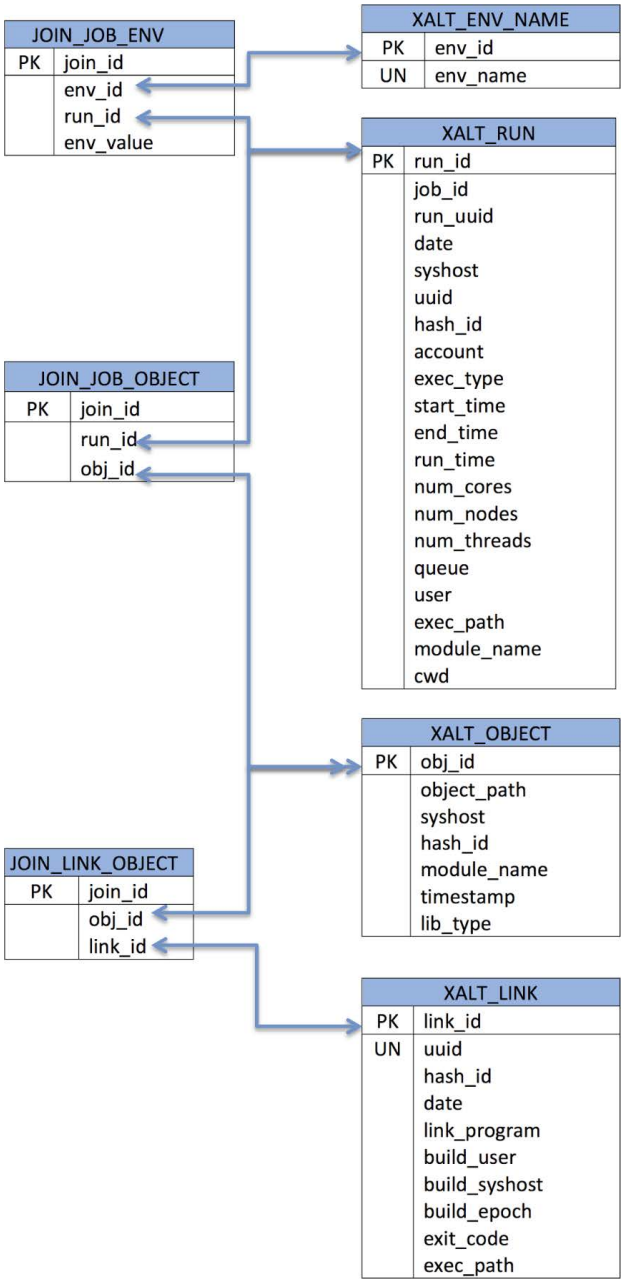

Figure 1: XALT **database schema.**

name, while the value itself is found in the `join_run_env` table in the `env_value` field.

All the information is interlaced with the help of UUID[3]. A UUID is generated when an executable is compiled. This identifier (UUID) is placed with the generated assembly code in the section header of the user's executable. At runtime, the code launcher wrapper parses the executable to get the section header details including the UUID, which provides a way to link the runtime entry with the corresponding entry in the link table.

It is fair to ask why we store the shared libraries twice, once at link time and another at runtime. This is because versions of shared libraries used by a program can change over time. When a program is built, it links with the libraries that are present at the time. In the future, there may be a newer shared library that is used instead and this should be tracked.

[3]The universally unique identifier is a 128-bit value used to uniquely identify information.

Listing 1: SQL query for compiler usage

```
SELECT link_program, count(*) as count
  FROM xalt_link
 WHERE build_syshost = [syshost]
 GROUP BY link_program ORDER BY count desc
```

3. USAGE REPORTS

In this section we present results taken from the XALT database at the National Institute of Computational Sciences (NICS) covering a period from October 2014 through June 2015. This data is from a Cray XC30 supercomputer called Darter [2].

Results similar to the ones shown here can be obtained directly from any XALT database and installation using scripts available in the XALT GitHub repository [4].

3.1 Compiler Usage

XALT stores the "link program" which makes it trivial to associate a link event with the calling program. Most of the time, the link program is the compiler itself. The storing of link program is an improvement of XALT over its predecessor ALTD [3], in which figuring out the compiler used to link an executable was a non-trivial task.

Figure 2 shows the number of times each link program calls the linker. This data can essentially be obtained by a simple MySQL query shown in Listing 1, where `[syshost]` should be substituted by the system name of interest. A more user-friendly python script (`compiler_usage.py`) wrapping this MySQL query that takes extra arguments such as date range and syshost is available from the XALT repository for anyone to use.

In Figure 2, the top nine linking programs are obviously compilers, where in aggregate the GNU compilers (`g++`, `gcc`, `gfortran`) top the list at 71% usage, followed by the Intel compilers (`ifort`, `icc`, `icpc`) at 22% usage, and the Cray compilers (`ftn_driver`, `driver.cc`, `driver.CC`) at 6% usage. For these results we have lumped the link program `c++` with `g++` since on Darter it is just a soft link to `g++`. In this plot, "other" makes up 0.3% total and comprises of: `make`, `bash`, `configure`, `build`, `ksh`, and `csh`. These are a small number of cases where the linker (`ld`) was called directly by either a Makefile, configure script, or shell.

Another interesting interpretation from this data is on programming language usage. If we make a relatively safe association of the compiler (i.e. link program) with the primary programming language of the program, we can then infer that about 45% of programs linked on Darter is written primarily in C++, 35% in C, and 20% in Fortran. Note however that we do not have any data to infer the size of the programs (e.g. number of lines, computational complexities, etc).

The compiler usage from counting linking instances in Fig-

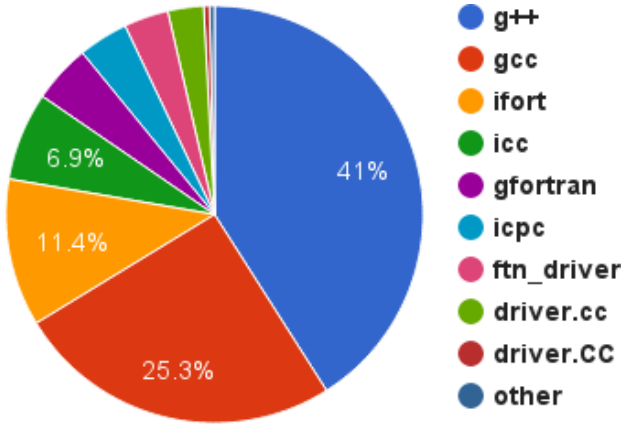

Figure 2: Usage of link program from 105,231 number of links. The GNU compilers (`g++`, `gcc`, `gfortran`) make up about 71% instances in aggregate, followed by the Intel compilers (`ifort`, `icc`, `icpc`) at 22%, and the Cray Compilers (`ftn_driver`, `driver.cc`, `driver.CC`) at 6%.

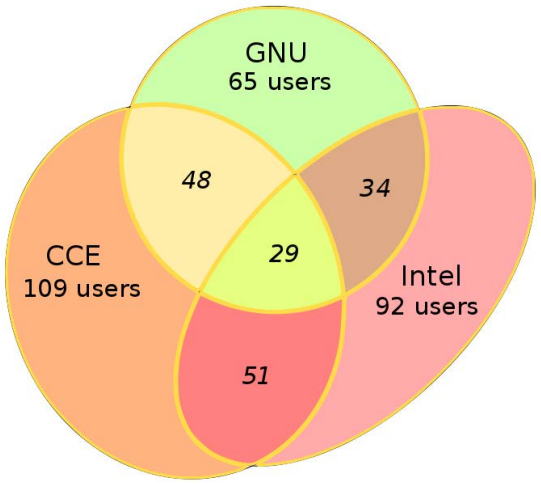

Figure 3: Area-proportional Venn diagram of the number of unique users for compilers on the system. Italics numbers show the subset of the unique users who have used different compilers. For example, 48 users out of 65 the GNU compilers and 109 Cray compilers users have used both compilers. 29 unique users have used all the compilers on the system.

ure 2 can be compared to Figure 3 where we show the number of unique users for each compiler. Although GNU compilers are used the most for linking, there are more unique users of Cray (CCE) and Intel compilers. Figure 3 also shows the number of users who have tried different compilers on the system, and that some users have only exclusively used the compiler of their choice.

One can also try to dig a bit deeper into this data to get more insight on how users choose the compilers they use. For example, a question that can be posed might be "is there a way to tell if someone used a compiler once (or a little bit) and then gave up and went to another compiler?" A 0^{th}-order approximation to answer this question is by looking at the usage ratio of the compilers each user uses.

Figure 4 plots this data for 30 unique users (out of 75, for brevity) on Darter who have used more than one compiler. For each user, represented by a number, the fractions of their use of CCE, GNU, or Intel compilers are shown as stacked bar chart. One particular example is user 6. From the plot we may speculate that user 6 tried CCE, and then decided to use GNU instead. Similarly, user 30 favors Intel compiler better after trying out CCE for a little bit. User 27 however chose to continue using CCE after trying out the other two compilers. This data of course does not give us enough information to tell us the chronological order by which users try out the compilers. We nevertheless may speculate that CCE is their first try since it is the default on Darter. If and when a more thorough and precise analysis is needed however, the XALT database can be interrogated to show the chronology of a particular user compiler usage, including information such as if the user has tried to link the same executable multiple times using different compilers. The latter is an indicator of user testing out the capability or performance of the different compilers.

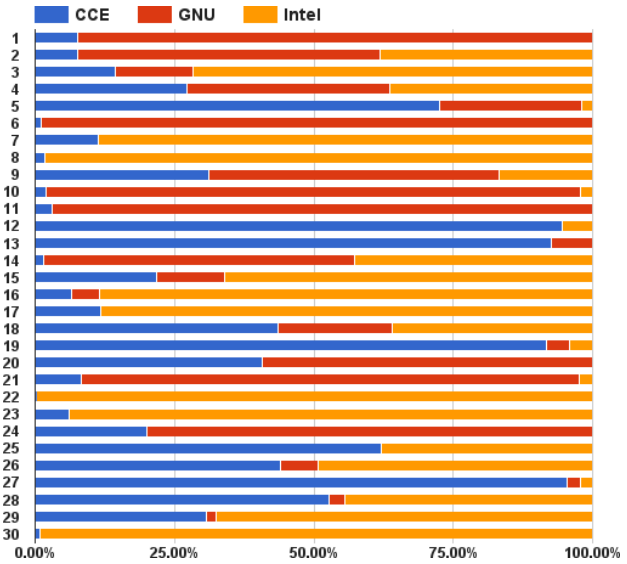

Figure 4: Usage ratio of different compilers by 30 unique users who have used more than one compiler on Darter.

3.2 The Most Used Libraries

There are obviously many definitions of "the most used", for example, by the number of distinct linkings, by the number of unique users, by the number of time a particular exe-

cutable using that library is launched, etc. For our purpose, we will look at the first two definitions of "the most used". By knowing which libraries are used most, an HPC center can devote its resources to make sure that these libraries have excellent support.

Even with this narrowed down definitions, one needs to filter out the non-interesting libraries and object files that most (if not all) programs link to such as the bootstrapping libraries (e.g. `crtX.o`), the C library (`libc.a`), the compiler-related libraries (e.g. `libgcc_*`) and the MPI libraries[4]. We make a further assumption that the most popular libraries are already provided by either vendor or the center's staff, and therefore have a modulefile associated with them. This assumption lets us group the most used libraries by the module name. Using the module name also lets us attribute the same library in the event that the actual object path changes over time, provided that an accurate ReverseMap[5] file is maintained.

Figure 5 shows bar charts of the most used libraries on Darter. The number of distinct linkings grouped by module name, and the number of unique users of the libraries are plotted. (Data used on this figure can be obtained directly using the contributed python script `library_usage.py` in the XALT repository.) In this plot, we have aggregated multiple versions of the same library to a single entry. We have also excluded the library for Cray Abnormal Termination Processing (ATP) to avoid skewing results, which is linked by default to all programs. At the top of the list is the Cray scientific libraries (cray-libsci) which provides common numerical libraries such as BLAS and LAPACK and is loaded by default. However, it is worth reiterating that although the cray-libsci libraries are included in the link line by default, it is only recorded by XALT in `xalt_link` table if it is actually used and linked to the the application (see §2). The next two most used libraries by the number of link instances are I/O libraries: HDF5 and NetCDF. However, if ranked by the number of unique users, then FFTW is third most used rather than NetCDF.

The number of unique users of a library may give better insight of the library usage because it takes out development cycles or profiling cycles for that library. In Figure 5 we see that the trends of library usage based on number of link instances and unique user are fairly similar, except in few number of cases such as FFTW, IOBUF, FSL, and GlobalArrays.

3.3 Top Executables

Data regarding what and how executables occupy the system resources is of great import to an HPC center. This kind of information is often needed for reporting to stakeholders and funding agencies. As described in §2, XALT tracks executable run by the parallel job launcher and is therefore suitable to produce information on jobs occupying the system resources. Although HPC batch systems and resource managers (e.g. SLURM, PBS, Moab) may also track this data for accounting purposes, XALT has further advantages over them. First, by wrapping the job launcher, XALT produces

[4]The filtering is only done for the reporting. The usage of these libraries are still recorded in XALT database.

[5]The ReverseMap is an optional feature of XALT that maps libraries to modulefiles. To generate the ReverseMap, the Lmod tool is required, and scripts are provided in the XALT repository to generate it.

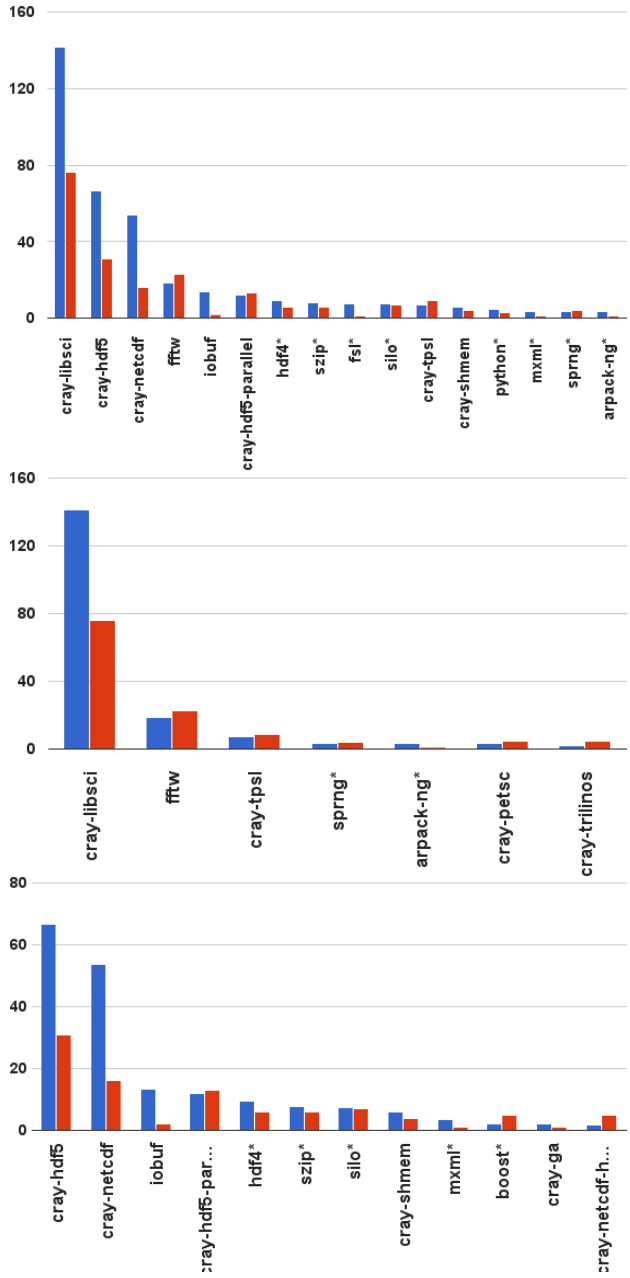

Figure 5: The most used libraries group by module name. The number of instances (blue) and the number of distinct users (red) are shown, where the number of instances (blue) has been scaled down by a factor of 100. On the top most panel the overall results out of over 57,000 total instances are shown. On the two lower panels we have split the results to show usage of numerical libraries (middle panel), and programming and I/O libraries (bottom panel) which make up about 50% the overall usage each. The asterisk in the legend indicates that the module is provided by the center staff while the rest are provided by the vendor.

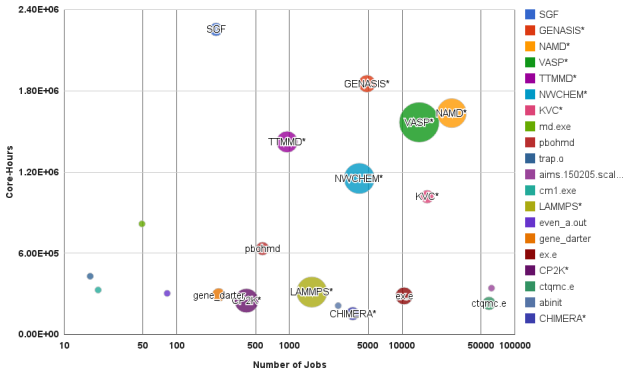

Figure 6: The top 20 executables consuming most time on Darter for during the specified time period. The vertical axis plots the core-hours, while the horizontal axis specify the total number of jobs (in log scale) for that executable. The ratio of these two numbers represents the average core-hours per submission. The bubble size represent the number of unique users that launched that particular executable.

Listing 2: SQL query for users of specified software

```
SELECT distinct(build_user) as user,
       max(date) as last_link
  FROM xalt_link xl, join_link_object jl,
       xalt_object xo
 WHERE xl.link_id = jl.link_id
   AND jl.obj_id = xo.obj_id
   AND xo.module_name like 'silo%'
 GROUP BY user
```

user	last_link
4d02fc0111b731d7	2015-05-29 12:47:16
08b74fac05d38427	2015-05-21 21:09:02
26df635d19291287	2015-03-26 13:38:16
0ee86a5a6cac658b	2014-10-22 21:35:09
792bcf860ad46ef2	2014-12-01 17:54:56
522b80ff02a97b88	2015-05-29 10:24:37
26c785a10116e105	2014-10-31 11:40:13

finer grain information that can then be correlated with the batch system record to describe how much time is actually spent in the parallel job versus the entire job script. Second, XALT tracks the actual number of compute cores used in the parallel job rather than requested by the job scripts (as usually recorded by the batch system) by parsing the arguments to the parallel launcher. XALT also tracks how these cores are used (i.e. by OpenMP or MPI) by recording and parsing the environment variables. Lastly, and perhaps most importantly, since XALT tracks the linking phase, it can present information on how the launched executable was build, including the compiler and libraries linked to it, providing the provenance of the executable (c.f. §5.1).

In Figure 6 we present some data from executables run on Darter during the specified time period. In this figure, we have also combined known executables that may reside in different filesystem paths into a common name. For example, the executable `namd2` built by different users, and therefore in different directories, would be recorded as different unique executables in the XALT database. For this particular report however we have combined them all into `NAMD*`. We also show the number of jobs for every executable, and the number of unique users of the executable. A python script to generate this report called `executable_usage.py` is available from the XALT repository.

3.4 Software Pruning

Maintaining multiple versions of a library or software is often both time and space consuming. Since both are finite resources, HPC centers must optimize their utilization in term of software support. But when is it time to remove older software version? How do we decide to prune rarely used version or even remove the software completely from the system, and thus free up both disk space and personnel time to support it? The answers to these question often depend on the usage of such software.

XALT can and has been used to answer those questions. There are typically two scenarios for software removal and the follow on necessary actions. Here we show how XALT can be useful in both scenarios.

In one scenario, a decision was made to remove a specific software. The HPC user support personnel then must make sure that users are aware of this decision and migrate their usage of the software to either a newer version or to a different, compatible software. Here rather than contacting all center's users, XALT can provide information to target only the users who have used that particular software in the past.

Listing 2 shows a MySQL query to obtain all the users who have linked against libraries provided by a particular modulefile. For illustration purposes here we picked a I/O library called "silo". The last time the linking was done by each user is also obtained. For results shown in this paper, we have hashed the username using MySQL `PASSWORD()` function to maintain anonymity. If one chooses to instead get results for a specific modulefile version, one may simply replace the pattern matching `LIKE` clause with the appropriate comparison (e.g equality) operator.

In the second scenario of software pruning, one typically does a survey of the current software available on the system and looks for the ones that either have never been used by users, or have fallen to disuse from obsolescence. Without something like XALT the task to determine the software disuse is difficult or nearly impossible, and relies mainly on anecdotal evidence. XALT makes it possible to have a data-driven decision for this process.

From XALT database, one can get the last time every modulefile (and the libraries or object files associated with the modulefile) is used and linked to an executable. One can also obtain the list of users who last linked to that software. Modulefiles or libraries that have not been used for a long time are candidates for pruning. One complicating point should be noted however. Since XALT only records object files *when* used in a linking event, XALT would not have any record for a library or object file against which it have never been linked. For these cases we turn to the `ReverseMap` file, since it should contain all the modulefile available on the

system. If a modulefile is in the `ReverseMap` file but not in the XALT database, then the libraries and object files provided by that modulefile have never been used.

A contributed script `software_lastusage.py` in the XALT repository does exactly this kind of last usage reporting. Listing 3 shows an excerpt of this report on Darter, where some omissions have been done both for brevity and to illustrate the more interesting data. (In this listing we have also hashed the username to maintain anonymity.) The top part of the listing shows libraries and the bottom part shows executables.

From the library report, one can immediately make the following observations: there is a lack of usage for ADIOS and AZTEC libraries; nobody has used the complex version PETSc so far; older versions of Trilinos were never used; all versions of Cray Libsci are still in use. From these observations, two actions that may be done include removing the libraries that are not used and encourage users to use newer version of Cray Libsci, which may contain bug fixes and performance improvements.

From the executable report, similar observations can be made: older versions of Gromacs are not really needed on the system, a variant build of VASP is never used and likely not needed, and some users still run older version of LAMMPS.

Listing 3: Excerpts from last usage report on Darter

```
%> python software_lastusage.py darter
```

Module Name	Last Linked	By User
adios/1.6.0	2014-10-10	522b80ff02
adios/1.7.0		N/A
...		
aztec/2.1		N/A
boost/1.55.0	2015-08-08	760722b91f
boost/1.56.0	2015-08-04	37a5932875
bzip2/1.0.6	2015-07-30	44ce9d260f
cce/8.1.8	2014-10-17	5e7dfeee49
cce/8.1.9		N/A
...		
cray-libsci/13.0.0	2015-05-27	0e7138df17
cray-libsci/13.0.1	2015-04-17	63078f4561
cray-libsci/13.0.3	2015-08-11	3235892916
...		
cray-petsc-complex/3.5.1.0		N/A
cray-petsc-complex/3.5.2.0		N/A
cray-petsc-complex/3.5.2.1		N/A
cray-petsc/3.3.06		N/A
cray-petsc/3.4.2.0		N/A
cray-petsc/3.4.3.0		N/A
cray-petsc/3.4.3.1	2014-10-21	4d02fc0111
cray-petsc/3.4.4.0	2015-08-10	0e7138df17
cray-petsc/3.5.1.0	2015-04-07	6b781fcc2f
cray-petsc/3.5.2.0		N/A
cray-petsc/3.5.2.1	2015-06-19	6b781fcc2f
...		
cray-tpsl/1.3.04		N/A
cray-tpsl/1.4.0	2014-10-21	4d02fc0111
cray-tpsl/1.4.1	2015-04-01	0e7138df17
cray-tpsl/1.4.3	2015-08-10	0e7138df17
cray-trilinos/11.10.1.0		N/A
cray-trilinos/11.12.1.0		N/A
cray-trilinos/11.8.1.0	2015-04-10	6b781fcc2f
...		
hdf4/4.2.10	2015-08-07	44ce9d260f
hdf4/4.2.9	2014-10-21	52c4e54c54

Module Name	Last Run	By User	Job ID
cp2k/2.5.1	2015-02-26	1005101a2c	398785
...			
gromacs/4.6.5		N/A	N/A
gromacs/4.6.7		N/A	N/A
gromacs/5.0.1	2015-08-10	11ab331a61	610893
...			
lammps/1Feb14	2015-06-17	78c321c666	580244
lammps/28Jun14	2015-08-11	660270a02a	612677
...			
vasp/4.6.36_tst		N/A	N/A
vasp/5.3.5	2015-08-11	673d6ecc38	612431
vasp/5.3.5_tst	2015-08-08	5fb1c32929	609648
vasp/5.3.5_tst_nc	2015-08-11	7044fe3d5a	612718
vasp/5.3.5_wannier90	2015-08-11	1c19e5801d	612545

4. USER COMMUNITY USAGE

4.1 TACC

XALT has been running and collecting data on the Stampede supercomputer at the Texas Advanced Computing Center (TACC) [9] since near its inception. The top panel in Figure 7 illustrates the compiler usage on TACC for a period of 90 days. From this figure we surmise that a lion's share of the programs build on Stampede are written in C and C++. We can also infer that the majority of all C, C++, and Fortran programs were compiled and linked with the Intel compiler.

Figure 8 shows some of the executables that consume the most CPU time on Stampede over the same period of time. A large amount of time was utilized by one user's custom code (as evidenced by a single unique user for that executable), while community codes such as LAMMPS, NAMD, VASP, and GROMACS also utilized considerable resources.

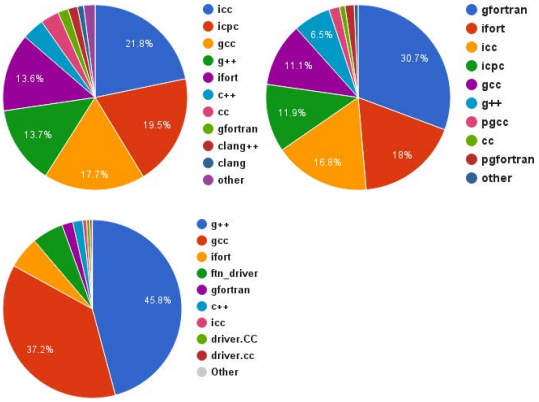

Figure 7: Usage of link program at TACC (top left), FSU (top right), and KAUST (bottom left). The number of recorded instances are approximately over 625000 (TACC), 19,000 (FSU), and 57,000 (KAUST).

4.2 Florida State University

As a supercomputing center managed by a relatively small staff, Florida State University (FSU) Research Computing Center (RCC) [5] was looking for a tool that generates accurate information about the compiler, library, and other software usage by its users. This information is essential to allocate enough resources towards the most used applications. XALT was installed in FSU HPC cluster mainly for collecting library and module usage statistics.

FSU uses an MySQL query similar to the one described in §3.1 to get compiler usage on their system, as shown in bottom left of Figure 7. This data covers a period of approximately 4 months. Interestingly, in contrast to the data from NICS, KAUST, and TACC, at FSU the Fortran compilers make up a larger share of the linking program here, followed by the C and C++ compilers.

FSU also monitors the modules used at runtime using MySQL query shown in Figure 9. Other MySQL queries that are regularly used to extract various statistic from XALT database are described in Listing 4

FSU has other accounting tools (Moab/Torque and SLURM

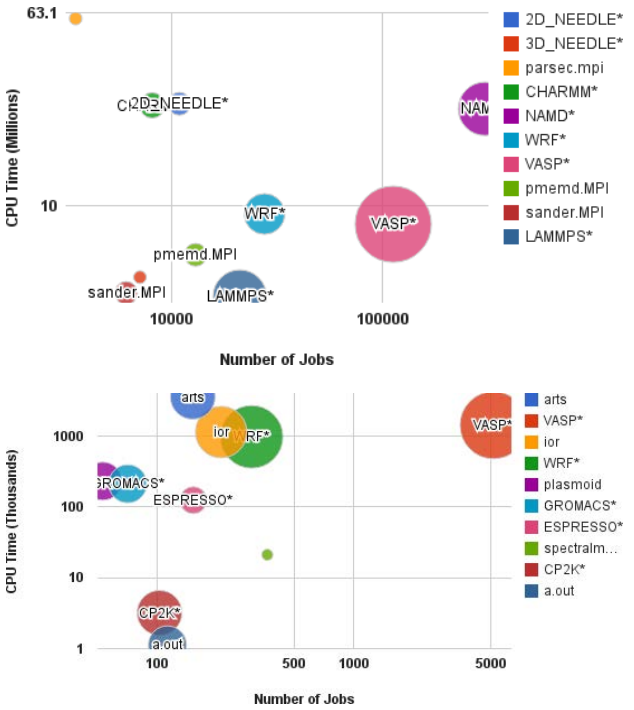

Figure 8: The top ten executables run on TACC's Stampede (top) and KAUST's Shaheen II (bottom). The data for TACC is for a 90 days period of a regular production stage, while the data from KAUST is from mid-May to mid-July and covers an acceptance test period.

```
SELECT xalt_object.module_name,
       count(date) AS jobs
  FROM xalt_run, join_run_object, xalt_object
 WHERE xalt_object.module_name is NOT NULL
   AND xalt_run.run_id = join_run_object.run_id
   AND join_run_object.obj_id = xalt_object.obj_id
 GROUP BY xalt_object.module_name
 ORDER BY Jobs desc;
```

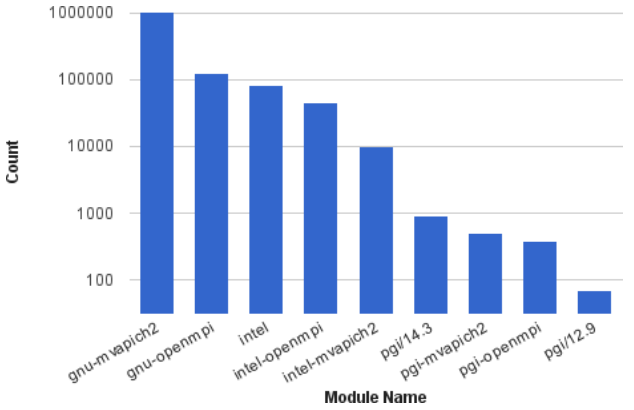

Figure 9: MySQL query and the resulting data for runtime module usage at FSU Research Computing Center over 4 months period.

Listing 4: Common MySQL queries used regularly at FSU Research Computing Center

```
#-- Which libraries are used at runtime ?
SELECT object_path, count(*)
  FROM xalt_object, xalt_run, join_run_object
 WHERE (object_path like '%.so%'
        or object_path like '%.a%')
   AND (join_run_object.run_id = xalt_run.run_id
        AND join_run_object.obj_id
            = xalt_object.obj_id)
 GROUP BY object_path ORDER BY count(*) desc;

#-- What libraries are used for compiling ?
SELECT object_path, count(*)
  FROM xalt_object
 GROUP BY object_path ORDER BY count(*) desc;

#-- Which (login) nodes are used for compiling ?
SELECT build_syshost, count(*)
  FROM xalt_link
 GROUP BY build_syshost ORDER BY count(*) desc;
```

+ MySQL + custom scripts) to analyze HPC usage per user/group and therefore do not use these information gathered by XALT. Currently, they are in the process of developing tools to do data mining to understand what type of jobs are run on the HPC cluster based on job name. The basic idea is to break the job name into substrings and calculate the occurrence of each substring along with the total CPU time used.

4.3 KAUST

At the King Abdullah University of Science and Technology (KAUST) [8], information about usage on their 16 rack IBM Blue Gene/P (Shaheen I) was limited to the workload manager project utilization and the feedback from users. The precursor project to XALT, ALTD, was put into production on Shaheen I in 2013 and analysis of the information collected after 6 months about the applications running on the system assisted in the design of benchmarks for the Shaheen II acquisition by selecting the most used libraries and applications.

The newly procured Shaheen II XC40 system [6] was installed in 2015 and XALT was put into production during the availability period of the site acceptance. During this time, KAUST staff, Cray engineers and selected users were given access to the machine. As of mid-July, grand challenge users and about 35 other users have been actively using the system. Figure 7 and 8 show compiler usage and top executables on the system as recorded by XALT during this time period. The data were obtained using query and python scripts described in §3.

On Shaheen II, the GNU compilers dominates the usage on the system, comprising over 70% usage. The rest are shared almost equally between the Intel compilers and the Cray compilers (CCE). Most of the the top ten executables run on the system over this period of time are community codes and were run by multiple users. This is in large part due to the acceptance testing phase done by the KAUST staff utilizing these codes.

As shown in Figure 10, KAUST HPC staffs have developed an early version of an automatic testing tool for the non-regression testing of hardware components and software updates. This is needed to detect errors and/or performance degradation, allowing the center to monitor issues such as reproducibility, and is managed through a continuous integration of Jenkins server [7]. The design of the regression test correlates the frequency of tested application with their effective use by the users. Mining the most used applications and libraries are easily doable thanks to XALT database. The staff can then prioritize these applications and libraries testing during a given upgrade of specific OS drivers to make sure they give consistent results.

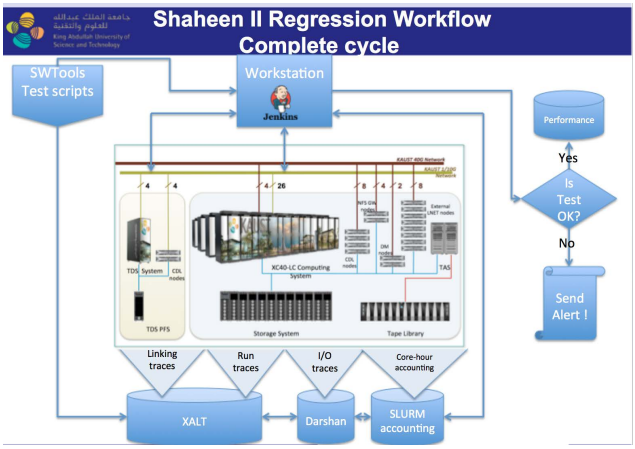

Figure 10: Early version of a new tool at KAUST incorporating XALT to do automatic regression testing of hardware components and software updates.

5. NEW FUNCTIONALITY

5.1 User Software Provenance

Reproducibility of results is one main tenet of scientific method. As computer software and computer simulation are now undeniably integral parts of scientific research, reproducibility of research conducted through software and simulation often means that we understand the history and origin, i.e. the "provenance", of the software used in such research.

How may XALT be useful for such thing? In this section we illustrate and make the case for using XALT as a tool for user software provenance.

One starts with a list of executables associated with job submissions by a user over a period of time. The list of unique executables by that user is simple enough to obtain via the following query

```
SELECT uuid, COUNT(1) as n_jobs,
       SUBSTRING_INDEX(exec_path, '/', -1)
         as exe
  FROM xalt_run
 WHERE user = '[username]'
 GROUP BY uuid
 ORDER BY n_jobs DESC
```

This query displays a unique list of executable's UUIDs and the number of jobs associated with that particular executable. (Note that XALT considers every linking as a unique executable, therefore two linking instances of executable with the same name are considered distinct.) The user then can be presented with the list of jobs with their respective start and end time for every unique executable's UUID. An example for a single UUID is trivial to do with the following query

```
SELECT run_id, job_id, date,
       FROM_UNIXTIME(start_time),
       FROM_UNIXTIME(end_time)
  FROM xalt_run
 WHERE uuid = '[UUID]'
```

while with a list of UUIDs this is easier to do with a scripting language iterating the UUIDs in the list.

Once the user determines which executable they would like to inspect, XALT can query the objects and modulefiles used to link such executable given the UUID. The following query obtains the object, modulefile, and link program used to build an executable:

```
SELECT xo.object_path, xo.module_name
  FROM xalt_object xo, join_link_object jo,
       xalt_link xl
 WHERE xo.obj_id = jo.obj_id
   AND jo.link_id = xl.link_id
   AND xl.uuid = '[UUID]'
```

One can also reconstruct the runtime environments for that executable given either the `run_id` or `job_id` (as given by the scheduler). Two distinct runtime enviroments are available. The first one is the runtime environment variables, which can be obtained with the following query:

```
SELECT env_name, env_value
  FROM xalt_env_name xe, join_run_env jr,
       xalt_run xr
 WHERE xe.env_id = jr.env_id
   AND xr.run_id = jr.run_id
   AND xr.job_id = '[JOB_ID]'
```

The second one is the object files loaded at runtime (and any modulefiles associated with them), which can be obtained with the following query:

```
SELECT object_path, module_name
  FROM xalt_object xo, join_run_object jr,
       xalt_run xr
 WHERE xo.obj_id = jr.obj_id
   AND xr.run_id = jr.run_id
   AND xr.job_id = '[JOB_ID]'
```

This latter one is especially important for dynamically linked executables for which XALT can identify the shared libraries loaded at runtime. For these executables, if the runtime loaded libraries are of different versions than the link time, undesireable behavior may occur. A change in software environment (e.g. the default loaded modulefile version) may also result in different versions of the libraries being loaded for the same executable run at different times. By tracking these runtime libraries, XALT gives the user a full picture of how an executable was built and run. This provides better software provenance for reliability, reproducibility, and error detection of software on our system.

A graphical and textual interface that combines all the elements software provenance discussed here is being developed to give user an easy and seamless access to XALT data.

The interface will allow the user to interrogate the build and runtime environments for every executable they run, including executables built by other users or HPC support staff in the case of community codes. Eventually this interface will be available from the XALT repository.

5.2 Anonymization

As part of our deliverables on our grant to NSF, and as a community driven need for sharing data, we can now produce files that contain anonymized data for researchers to study how users use our systems. Researchers who study the use of open source software might use this data to see how different fields of science use which open source software and how it evolves over time.

At one month intervals we produce a JSON file that contains a record of every parallel program that was run on our supercomputer for that month. In order to protect personal data about the jobs we substitute program names and directories into "secret" names. For example a user account named "smith" might become "U01254733" and the allocation might become "A05376861". We guarantee that the same user and allocation map to the same string. All user path information is also protected.

The program names get special treatment. If the user is running a community program like WRF or VASP, then it will be reported as such. Otherwise the program name is a 40 character hash of the name of the program, not the absolute path. In this way, different users executing "a.out" will still produce the same hash value.

A example of one job entry is shown in Figure 11. This example does not show all the shared libraries this program used for space reasons.


```
{
    "allocation": "A00084719",
    "date": "2015-03-02 00:20:26",
    "exec_path": "PARSEC*",
    "field_of_science": "Astronomy",
    "host": "stampede",
    "job_id": "4922626",
    "linkA": [
        {
            "library_module_name": null,
            "library_path": "/lib64/libpthread-2.12.so"
        },
        {
            "library_module_name": null,
            "library_path": "/lib64/librt-2.12.so"
        },
        {
            "library_module_name": "intel/13.1.1.163",
            "library_path": "/opt/apps/intel/13/composer_xe_2013.3.163/
                  mkl/lib/intel64/libmkl_core.so"
        },
        {
            "library_module_name": "intel/13.1.1.163",
            "library_path": "/opt/apps/intel/13/composer_xe_2013.3.163/
                  mkl/lib/intel64/libmkl_intel_lp64.so"
        },
        {
            "library_module_name": null,
            "library_path": "/opt/apps/intel/13/composer_xe_2013.3.163/
                  mkl/lib/intel64/libmkl_sequential.so"
        },
        {
            "library_module_name": "mvapich2/1.9a2(intel/13.0.079)",
            "library_path": "/opt/apps/intel13/mvapich2/1.9/lib/libmpich.so.8.0.1"
        },
        {
            "library_module_name": null,
            "library_path": "/opt/ofed/lib64/libibmad.so.5.2.2"
        },
    ],
    "module_name": null,
    "num_cores": 16,
    "num_nodes": 1,
    "num_threads": 1,
    "run_time": 1052.0,
    "start_time": 1425277226.45,
    "user": "U00361810"
},
```


Figure 11: Job information that has been anonymized. Specifically, "allocation", "exec_path", and "user" have been anonymized.

6. CONCLUSIONS

XALT has been in production for over a year now and in use at several sites. We presented data from a subset of these sites and showed some of the analyses that can be done, and some of the ways XALT is being extended. This includes identifying the most used libraries and executables, software pruning, preliminary insight into compiler abandonment, and even extending into regression testing.

New functionality in XALT was also presented - the ablity to anonymize data and early work in providing access to provenance data.

Acknowledgment

We would like to generally thank the XALT community in order to acknowledge the great feedback and bug reports.

This work was supported by the NSF award 1339690 entitled "Collaborative Research: SI2-SSE: XALT: Understanding the Software Needs of High End Computer Users."

This material is based upon work performed using computational resources provided by the University of Tennessee's Joint Institute for Computational Sciences and the Texas Advanced Computing Center (TACC) at the University of Texas at Austin.

7. REFERENCES

[1] K. Agrawal, M. Fahey, R. McLay, and D. James. User environment tracking and problem dectection with XALT. In *Proceedings of the First Workshop on HPC Tools for User Support (HUST14) Workshop held in Conjunction with the International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*, HUST14, New Orleans, LA, November 2014.

[2] M. Fahey, R. Budiardja, L. Crosby, and S. McNally. Deploying Darter – a Cray XC30 system. In J. Kunkel, T. Ludwig, and H. Meuer, editors, *Lecture Notes in Computer Science*, ISC2014, LNCS 8488, pages 430–439. Springer International Publishing, Switzerland, 2014.

[3] M. Fahey, N. Jones, and B. Hadri. The Automatic Library Tracking Database. In *Proceedings of the 2010 Cray User Group*, CUG10, Edinburgh, May 2010.

[4] Fahey-McLay. XALT, 2015. https://github.com/Fahey-McLay/xalt.

[5] Florida State University Research Computing Center, 2015. http://rcc.fsu.edu.

[6] B. Hadri, S. Kortas, R. Feki, G. Khurram, and G. Newby. KAUST's Cray XC40 system Shaheen II. In *Proceedings of the 2015 Cray User Group*, CUG15, Chicago, IL, May 2015.

[7] Jenkins. An extensible open source continuous integration server, 2015. http://jenkins-ci.org.

[8] King Abdullah University of Science and Technology KAUST, 2015. kaust.edu.sa.

[9] Texas Advanced Computing Center, 2015. http://www.tacc.utexas.edu.