XALT Design and Installation Manual

version 0.5

Mark Fahey

Robert McLay

Kapil Agrawal

* much of sections 1 and 2 are taken from the HUST14 paper.

- Abstract
- XALT Introduction
- XALT Design
  - Description of XALT
  - Requirements
  - Key Assumptions
  - Wrappers
  - Portability aspects
  - Database
- Installation
  - Prerequisites
  - Unpacking
  - Decisions to be made first
  - Installation
    - [Optional] Install Lmod
    - Lmod installation
    - Install XALT
  - SETUP next steps
  - Testing
  - Other Transmission methods:
    - syslog
    - direct to database
  - Installation Considerations
    - Intercepting linkers and job launchers:
    - Env Var BlackList
  - Creating the Reverse Map
    - Examples
  - Known issues:
- Appendix A - Alternative Technologies
- References

Abstract

This work improves our understanding of individual users' software needs, then leverages that understanding to help stakeholders conduct business in a more efficient, effective, and systematic way. The product, XALT, builds on work that is already improving the user experience and enhancing support programs for thousands of users on twelve supercomputers across the United States and Europe. XALT is designed to track linkage and execution information for applications that are compiled and executed, respectively, on High Performance Computing (HPC) systems. In addition, XALT will compare the runtime environment (at execution) against the compile time environment (from linkage information) and provide users with information to resolve any runtime inconsistencies if appropriate. It will highlight the products our researchers need and do not need, and alert users and support staff to the root causes of software configuration issues as soon as the problems occur. A key objective of this work is generating the information needed to improve efficiency and effectiveness for an extensive community of stakeholders including users, sponsoring institutions, support organizations, and development teams. Efficiency, effectiveness, and responsible stewardship each require a clear picture of users' needs. XALT is an important step in the quest to achieve that clarity.

XALT Introduction

The world of computation is a challenging one: tight budgets, high demand, oversubscribed resources, increasing scope, and growing complexity. Given these realities, how can we make the most effective use of limited computing cycles and labor hours? During its five-year operational lifetime at the Texas Advanced Computing Center (TACC), the decommissioned Ranger supercomputer ran 2.69 million jobs and delivered more than two billion CPU-hours to 4,000 users from 359 institutions. It supported the computational needs of 2,244 separate open science research projects representing every imaginable research discipline from the hard sciences to humanities [1]. Statistics from other leadership-class systems paint a similar picture. To accommodate such scope and breadth, the largest supercomputers host a rich catalog of applications and libraries in a variety of configurations. The recently decommissioned Kraken Cray XT5 at the National Institute for Computational Sciences (NICS), for example, hosted more than 150 system-level applications and libraries [2]. Many would operate properly only with a particular choice of compiler, a specific Message Passing Interface (MPI) communications library, and other essential settings in the user's working environment (as is the case in other supercomputing environments.)

The associated costs and risks are significant, but so are the opportunities:

- Building, maintaining, and supporting a large software stack is a labor intensive endeavor requiring skilled practitioners. Is this time and money well spent? Do we

know how often research teams use each software product? XALT allows administrators and other support staff to consider demand when prioritizing what to install, support, and maintain.
- It is not unusual for users and consultants to consume long hours diagnosing an error caused by attempting to run an experiment with improperly configured software. Such problems derail scientific progress, and are particularly common among users transitioning to high end computing resources for the first time. XALT will flag jobs that require the attention of support staff, deliver alerts to users regarding the fundamental causes of problems preventing their jobs from running, and collect metrics that improve training, documentation, and outreach programs.
- Datasets, dashboards, and historical reports generated by XALT and the systems with which it interoperates will preserve institutional knowledge and lessons learned so that users, developers, and support staff do not have to reinvent the wheel when issues arise that have already been encountered.

In the executive summary of its final report, the NSF Task Force on Software for Science and Engineering, Advisory Committee for CyberInfrastructure, stated that there are no “generally accepted quantitative metrics for determining what software researchers most heavily use” [3]. The statement is one important observation about a larger issue. Our systems are complex in size and scope; we must manage them in a systematic way. The methodologies and tools for doing so, however, are still in their infancy.

On another front, the Yale report on the topic of “reproducible computational research” suggested “scientists make all details of the published computations (code and data) conveniently available to others” [4]. Much of what they suggest is highly commendable and more importantly very achievable. Complete documentation from the source code to versioning to programming and runtime environments to publishing of data would allow other researchers to reproduce everything about the researcher’s code and how it was built. Hardware and some parts of the software environment are outside the researchers control especially when running at a remote computing center, but rebuilding an application code with the exact same software and version could be accomplished with a chance for “very similar” results with only round-off error differences.

Computing centers can help researchers in a way that only they can fill. Computing centers at both the university and national level can provide an automatic way to collect the information on the versions of software used by the researcher. For example, NICS and TACC provide two similar but slightly different prototypes (ALTD [5] and Lariat [6], respectively) that capture the libraries and their versions used by each researcher for each code they build and run. This solves part of the documentation problem stated above; for example, NERSC uses ALTD [7] so that users can find out this provenance data from old builds with a simple database query, allowing researchers to build their codes exactly like they did months or years before. Our new effort XALT is under development to combine and extend the ALTD and Lariat infrastructures to capture even more information - basically almost everything mentioned

above. Collecting this information is relatively straightforward (as shown by the prototypes) and has been proven to be very effective. It would help the researchers greatly with providing the information the Yale report [4] recommends.

Lastly, there is also a security benefit from tracking all the software used by researchers. For example, by tracking the shared libraries required at link time and then again at run time, a center can track the use of these shared libraries by their user community and detect when changes have occurred over time that might point to performance differences or a hacked SSL library.

In summary, the data mining performed by XALT will help provide a detailed and accurate survey of the usage of software installed by vendors, staff, and users. By tracking software usage, user support quality and efficiency can be improved. Also, this can help build a community around analytics regarding software needs, trends, and issues at the level of the individual job. Every center should be doing this for a variety of reasons from better user support to provenance data collection to security related concerns.
This paper is organized as follows. Section II provides an overview on XALT, highlighting the requirements and the design of the project. Section III describes the methodology and implementation. Section IV presents results from early data mining efforts and summarizes expected analyses that will be performed. Section V presents related work done in this direction and how these works are insufficient for the goals of XALT. Section VI provides an insight on future work that will be carried on XALT. Lastly section VII provides a summary of XALT and its abilities.

XALT Design

XALT will collect accurate, detailed, and continuous job-level and link-time data and store that data in a database; all the data collection is transparent to the users. The data stored will 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 our researchers do and do not need. XALT will compare the run time environment (at execution) against the compile time environment (obtained from linkage information) and alert users and support staff to the root causes of software configuration issues as soon as they occur. XALT will help mitigate the difficulties new users encounter and identify opportunities to improve documentation, education and outreach programs.

Description of XALT

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. In addition, XALT will monitor and track individual code executions, which would in turn help get attention of

support staff and/or deliver alerts to users regarding the basic causes of problems preventing their jobs from running, and collect metrics that improve training, documentation, and outreach programs.

Our approach is based on wrappers that intercept both the GNU linker (ld) to get linkage information and the code launcher (like mpirun, aprun or ibrun) when the code is executed. For the initial release of the project a handful of architectures are targeted, though subsequent releases will include support for additional architectures (these may require supporting additional linkers and code launchers). Wrapping the linker and the code launcher is a clean and efficient way to intercept information automatically and transparently, as nearly every user will invoke the linker (ld) at compile time and launch the code through the code launcher (aprun, mpirun, ibrun).

XALT will track static, shared and dynamically linked libraries. XALT will not only track libraries linked into an application, but it will also (in a subsequent release) detect function calls that need to be resolved by external libraries. User defined functions and auxiliary functions in the external libraries will not be tracked. XALT will not track all function calls because that is not likely to increase the understanding of library usage.

XALT tracks libraries linked into an application and also detects function calls resolved by external libraries. User defined function calls are not stored, and any auxiliary functions in the external libraries are not stored. By storing compile time libraries, at runtime an application's runtime environment can be compared against the compile time environment and information can be provided to the user about what may need to be done, if anything, to fix runtime issues.

Our routine wrapper for the linker (ld) intercepts the user link line and parses the command line to capture the link line, which is then stored in a json file in a user's XALT directory. At the same time an ELF section header is included in the user's code, which is a marker that will be used to record any subsequent usage 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 some job-specific environment variables from the batch system, such as job id (PBS_JOBID in the case of PBS). It also detects dynamic libraries loaded during the runtime and stores this information in another json file. These json files (i.e. the files created each time a code is compiled and executed) is then later stored in a database by running a script. The data then be mined to provide reports, which can improve our understanding of library usage and many operations related questions can be answered. 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 will be automated and scheduled during non-peak hours so that there is no adverse impact on the user experience. The implementation and alternatives are described in the next section.

Please see **Appendix A** for a more detailed discussion of alternative technologies.

Requirements

The XALT requirements and design goals include:

- Avoid impacting the user experience: A primary design goal was that no matter what the XALT wrapper does (work or fail), 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. We do note that XALT actually links its own object file into the user executable and that alteration of the link line can in rare cases may require changes in the way the user links or launches code.
- Must work seamlessly on any cluster, workstation or high-end computer: With different stakeholders showing their interest in improving infrastructure, and doing research in a more effective, efficient and systematic way. We want XALT to be a practical and usable solution working on any cluster, workstation or high-end computer.
- Must support both static and dynamic libraries: XALT tracks libraries linked during the linking process, and detects both static and dynamic libraries. At runtime, XALT detects dynamic libraries that are loaded during the execution phase – in this way one can determine if an executable uses different versions of dynamic libraries over the course of its lifetime.
- Lightweight solution: One of the primary objective was to develop XALT as a lightweight solution with essentially no overhead at compilation and runtime.
- Alert users and support staff of software configuration issues: It is our goal to identify jobs that require support staff's attention; along with this it should deliver alerts to users regarding the fundamental causes of problem preventing jobs from running. XALT will not only guide support staff in maintaining and supporting a large software stack but will also reduce long hours spend in diagnosing errors of individual users and consultants.

Key Assumptions

In the design of XALT, a few assumptions were made that may or may not apply at other centers. These assumptions are now summarized:

- *More than one linker and/or job launcher to intercept:* It is often the case that clusters have many MPI installations (MPICH2, MVAPICH2, etc.) with multiple versions. Each such installation is likely to have its own launcher (e.g. mpirun). To be a viable solution on any cluster, we must be able to intercept each code launcher. If a site has more than one code launcher (apart from aprun, mpirun, ibrun), then wrappers for each might need to be provided. In this case the site staff can decide how they want to intercept them. This is discussed in more detail in the upcoming section.
- *Not everything on the original link line should be captured:* we only need libraries that are actually linked into the application: Often libraries are specified on the original link

line that are not actually linked into the application. The product must exclude anything that is not actually linked to the executable.

- *Track function calls:* Function calls that need to be resolved by external libraries will be tracked. User defined function calls, and any auxiliary functions in the external libraries will not be tracked. It is assumed that only the calls needing to be resolved by external libraries worth tracking, and that all the remaining functions and procedures would be of little interest.
- *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, NICS and TACC use module files to provide environment variables with paths to libraries and applications with version numbers embedded in the paths. This might make easy to extract version information directly from the path. However, this will be site specific.

Wrappers

Linker

The linker (ld) wrapper intercepts the user link line. Since in many cases more libraries are included on the link line than are actually used, we go through a four-step process to identify the libraries that are actually linked into the executable while at the same time including an ELF section header in the user's code. Below is the high-level representation of the steps:

- Generate assembly code: During this phase, the linker (ld) wrapper calls a python script to generate assembly code that is stored in the section header of the user's executable. The assembly code contains nine fields – XALT version (XALT_Version), build system host name (Build.Syshost), compiler (Build.Compiler; this data is collected from passing the PSTREE to the script and extracting the compiler information), OS (Build.OS), User (Build.User), a universally unique identifier associated with each build (Build.UUID), year (Build.Year), date (Build.Date) and epoch (Build.Epoch). These fields are necessary to accurately track the executable in the jobs table back to the correct machine, user who compiled it, and machine on which it was built. Figure 1 is an example of the assembly code. The details are written to an 'xalt.o' file in a user-level 'tmp' directory.

```
.section .xalt
.asciz "XALT_Link_Info"

.byte 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
.asciz "<XALT_Version>%%%%"
.asciz "<Build.Syshost>%%darter%%"
.asciz "<Build.compiler>%%driver.cc%%"
.asciz"<Build.OS>%%Linux_%_%_3.0.101-0.29-default%%"
```

```
.asciz "<Build.User>%%kagrawa1%%"
.asciz "<Build.UUID>%%bd97b98b-2169-416e-85c1-762be8846dd2%%"
.asciz "<Build.Year>%%2014%%"
.asciz
"<Build.date>%%Fri_%_%_Jul_%_%__%_%_4_%_%_13:37:01_%_%_2014%%"
.asciz "<Build.Epoch>%%1406914621.1%%"
.byte 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
.asciz "XALT_Link_Info_End"
```

Figure 1: XALT assembly code

- Generate link text: In this phase we use the tracemap output from the real linker and write all the libraries that were linked to the executable and the xalt.o file (containing the assembly code) into one text file (link.text), which will be later parsed to exclude some specific libraries which are of less interest.
- Generate link data: The main aim of this step is to refine the information collected in the above two steps. The linker (ld) wrapper calls a python script which parses the link text, removing all redundant information such as lines consisting of tokens like ':' or lines storing temporary object paths (/tmp/ccT33qQt.o). Once done all the relevant information is then dumped in one .json file in the user's '~/.xalt.d' directory.
- Transmit collected data in '.json' format: The current release supports three different methods to get the data into the database. By default, the data is transmitted to the a .json file in the user home directory, and then later asynchronously update the XALT database (e.g. daily or weekly as configured at the site) rather than hitting the database frequently with potential performance effects. However, as a portability feature, we provide alternate methods to get the data into the database; this is discussed in more details in the Portability aspects section.

Code Launcher

Launching a parallel job on compute nodes is often done via a batch system (like PBS, Slurm, or LoadLeveler) through a parallel job launcher such as aprun, mpirun, mpiexec, or ibrun. These job launchers are intercepted to provide a measure of application usage and a secondary measure of "library usage." XALT tracks both start and end time as well as parallel launcher options and environment variables that will prove invaluable in both debugging and job analytics to understand good/bad behavior. Below is the high-level representation of the steps:

- Find executable: Each job launcher wrapper (like aprun, mpirun, or ibrun) has its own python script to find executable, we need this so as to extract the information stored in the ELF section during the link time (by linker wrapper). This section in future will help us relate to the information collected by linker wrapper at compile time.
- Get actual launcher and command line option: Here we get information regarding actual launcher and its options (not all options); this would in turn be called once we are done with our wrapper.

- Collect link time, job, and shared libraries information and dump it to a file: The command objdump is run on the executable to extract link time information as stated earlier, along with this job information like unique run ID, start date/time, end date/time, et al. Additionally, the ldd command is run to extract information regarding shared libraries required by the executable at run time. All this information is dumped into a '.json' file in the user's '~/.xalt.d' directory.
- Upload .json file to XALT database: This is the same process described at the end of the section above on the Linker wrapper.

Portability aspects

One of the main objectives of this project was to design an infrastructure that works on any cluster, workstation or high-end supercomputer. There is a strong possibility that there is more than one linker and/or job launcher to support even at one site. For the initial release of the project we have targeted several architectures, and in subsequent releases we will include support for additional architectures as time permits and/or the community shares them. New architectures may require supporting additional linkers and code launchers.

In the alpha release, as described previously, we elected to dump '.json' files in the home directories of each user, which would then later be mined and processed by another utility. This design is portable yet may not be the desired choice for all sites. This process can be changed to meet an individual's site's requirements; discussed below are the three methods that will be provided in a subsequent release of XALT:

- Using Files: This is the default for XALT - all information is dumped into '.json' files (one each for compile time and runtime), then a script parses these files and uploads the data to the XALT database.
- Using SYSLOG: Some sites may see storing these files in a user's home directory as a security issue. Instead, we can write captured data directly in SYSLOG (though this requires breaking the data into parts because of size constraints) and subsequently retrieved by a script/job which can consolidate the data and write it into the XALT database.
- Direct Database Interaction: This feature is similar to what ALTD offers in that all the linkage and execution information is directly inserted into the XALT database in real time when a user compiles or executes a code.

Database

The XALT database has several tables. Since there are several many-to-many relationships, we have designed the database with "join" tables that reduce storing 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), a single entry is added to the XALT_LINK table. This table contains information about the program such as the builder, a time stamp, and what program linked it. There will be multiple object files and libraries that

constitute the built program. Each of these is stored by XALT_OBJECT. There is only one entry in the XALT_OBJECT for each object path that has the same sha1sum [8]. The join table named JOIN_LINK_OBJECT connects the XALT_LINK entry to the object files that belong to the program.

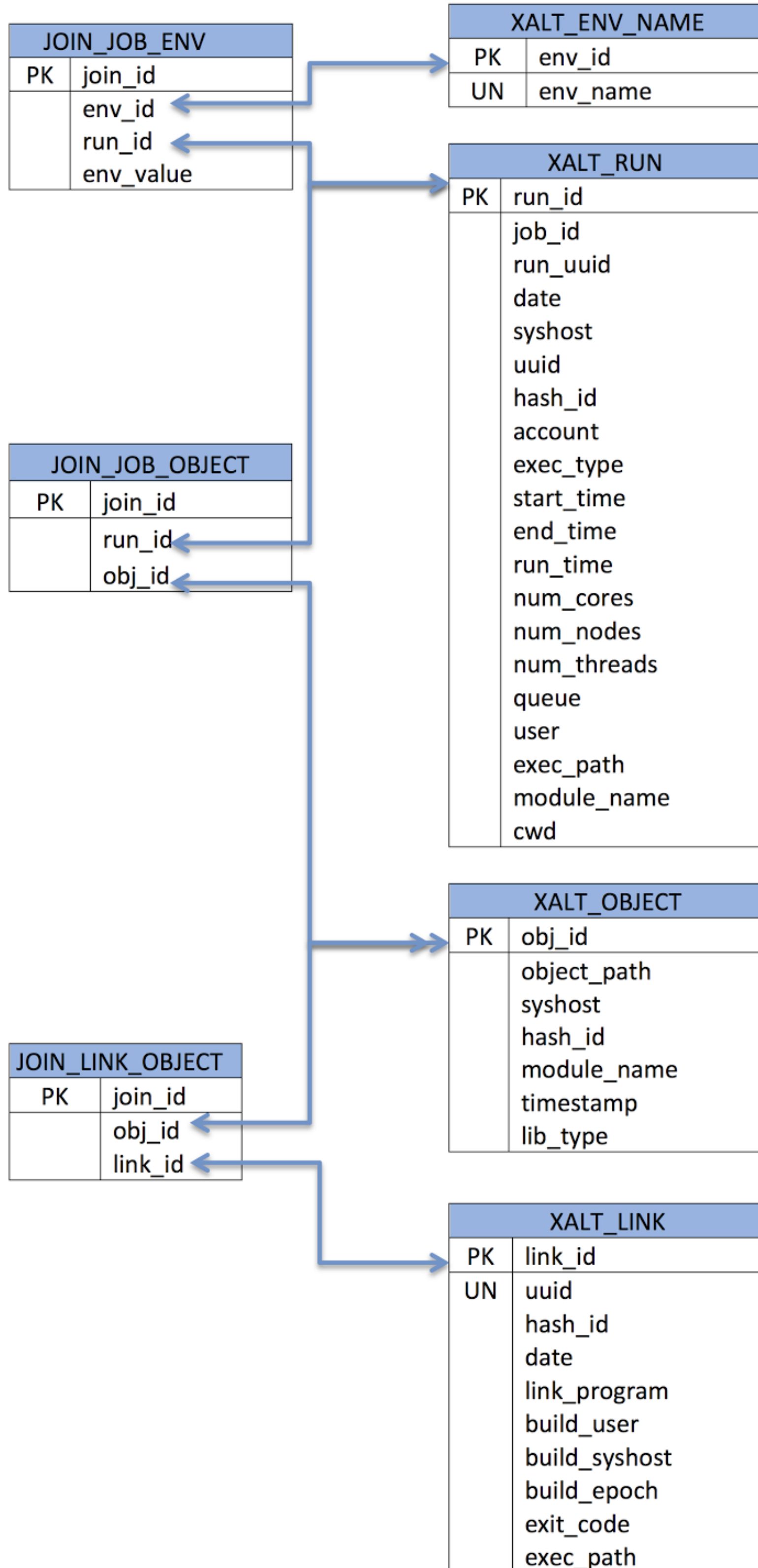
When a program is executed via a launcher, we create a single entry in the XALT_RUN. We take the shared libraries that belong to that executable and place them into the XALT_OBJECT table and connect to the XALT_RUN entry via the JOIN_RUN_OBJECT join table. We also record the Environment variables in the XALT_ENV_NAME table and use the JOIN_RUN_ENV table to connect the values of the variable to the name and the program execution.

Figure 2 shows basic layout of XALT database. All the information is interlaced with the help of uuid. A universally unique identifier (uuid) is generated when an executable is compiled. This identifier (uuid) is placed with the generated assembly code in section header of the users executable. At run time, the code launcher wrapper parses the executable to get the section header details including the uuid which provides a way to link the run time 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 run item. The reason is to track changes in versions of shared libraries automatically used by a program over time. When a program is built, it links with the libraries that exist at the current time. In the future, there may be a newer shared library that is used instead. We wish to track that.

Installation

This section describes the installation of XALT. Installation is a partly automatic and partly manual process at this time described below.



JOIN_JOB_ENV
PK
join_id
env_id
run_id
env_value
XALT_ENV_NAME
PK
env_id
UN
env_name
XALT_RUN
PK
run_id
job_id
run_uuid
date
syshost
uuid
hash_id
account
exec_type
start_time
end_time
run_time
num_cores
num_nodes
num_threads
queue
user
exec_path
module_name
cwd
JOIN_JOB_OBJECT
PK
join_id
run_id
obj_id
XALT_OBJECT
PK
obj_id
object_path
syshost
hash_id
module_name
timestamp
lib_type
JOIN_LINK_OBJECT
PK
join_id
obj_id
link_id
XALT_LINK
PK
link_id
UN
uuid
hash_id
date
link_program
build_user
build_syshost
build_epoch
exit_code
exec_path

Prerequisites

There are a few prerequisites for running XALT:

- For clients:
  - Python v2.6 or later
  - Python MySQLDb Module

- For servers:
  - MySQL with proper ip ranges opened for client machines

Unpacking

Choose an installation directory like /usr/local/xalt or /sw/xalt and untar xalt.tar into this installation directory. From now on, this directory is referred to as XALT_DIR. As shown in Figure 1, the library XALT is composed of several directories:

Decisions to be made first

There are several decisions to be made about how a site wants to install XALT. First, if a site has multiple machines, they have to decide if they want one database (one set of common tables) to hold all the information for all machines or if they want to have multiple database (typically one per machine). Both are valid installation options and easily set up in either case.

> **One database**: If you go with one database, then you *may* need a script per machine [samples provided in the distribution] to get the data into the database (depending on the method you choose),
>
> **Multiple databases**: In this scenario, a site just has to set up multiple databases (i.e., one server with multiple database like xalt-machine1 and xalt-machine2) You will need a script per machine to get the data into the database unless you are doing "direct to database." Note that if you are willing to edit the code directly, you could go with one database and multiple sets of tables per machine - not supported.

Regardless of the decision above, if your machines have different software installations (module lists) *AND* you want the ReverseMap support in XALT, then you will need a [Lmod] ReverseMap per machine which means having a build of Lmod per machine.

The next step is deciding where a few files are to be located; these files being the database xalt_db.conf file (mysql database access information) and the reverseMapD directory and reverseMap file(s). We suggest for simplicity that that go in XALT_DIR/etc, but this is up to the site to pick the location. It may be that the site wants the xalt_db.conf file somewhere more hidden/secure. This can be chosen with the configuration option --with-etcDir=ans or

overridden at runtime with XALT_ETC_DIR. Note that you will need to have a XALT_ETC_DIR directory for each machine.

Subsequently, the site needs to decide on whether they want to support the reverseMap functionality. The reverseMap is our code word for a file that has a mapping library paths or executables back to the appropriate modulefile (if it exists). This ability requires the creation of the module reverseMap, which is a result of running the "spider" utility from the Lmod [9] module system. One does not have to replace TCL module system with Lmod to get this functionality, it just needs to be installed

Furthermore, if a site has multiple code launchers or linkers, then the site has to decide how they want to intercept them. There are a variety of ways this can be accomplished, discussed below in the section on "*Intercepting linkers and job launchers.*" One possible way to do this is with Lmod in conjunction with XALT, but this requires *replacing* TCL modules with Lmod.

In summary, at this point the site needs to have made several decisions which will help guide the outline of installations steps next.

Installation

[Optional] Install Lmod

This section starts off with an installation guide for Lmod. Installation is optional, but if you want support for the reverseMap and/or you want to use Lmod in conjuction with XALT to intercept the linkers and launchers, then you'll want to install Lmod. For just reverseMap support, you do not need to replace TCL module system. To intercept the linkers and launchers and have XALT always first in the path, you'll want to replace TCL modules with Lmod.

Lmod will need to be built for each machine it is to be used on.

Lmod installation

1. Get Lmod-5.8rc2 or greater from…
    a. git clone git@github.com:TACC/Lmod.git or download Lmod-<ver>.tar.gz from github.
2. Pick a location to install like $SWBASE/lmod
3. Follow the INSTALL directions that are included with Lmod
4. Create local config files that users will source
5. Documentation for Lmod can be found at: https://www.tacc.utexas.edu/tacc-projects/lmod

For step 2, one consideration is whether or not to set up a system cache file. This is very useful if your system will have lots of modulefiles. You should set up a system cache file if

your users complain about how long it takes to do "module avail". This can happen with about 100 to 200 hundred modulefiles especially if they sit on a parallel file system like Lustre. If you set up a system cache file, then you have to update it every time new software is installed (e.g., run a cron job.) If you don't set up the system cache file, then lmod creates a cache file in the user's home directory, which means every so often the user sees a delay but not every time. Also, if you create a system cache file, y*ou need to think about the case wh*ere you have multiple systems sharing the same filesystem -- i.e., the cache file cannot be in the same place. Because Lmod needs to be built for each machine it will be run on, judicious organization of the installations could make this easy.

For step 2.d, we provide an example "source" file (lmod_startup) that need to be sourced at login to set up the Lmod module function - this can be done system wide or for just select users. We also provide a "system_name" script to be configured by the site that will help the startup script understand local naming conventions. For a testing phase, these files are very useful, but there are limitations (especially when one shell calls a different shell.)

Install XALT

Although it is possible to install XALT only once for several machines, we suggest installing XALT once for each machine. This simplifies the installation process and directory hierarchies.

1. git clone git@github.com:Fahey-McLay/xalt.git or download xalt-<ver>.tar.gz from github. At least version 0.5.0 or greater.
    a. XALT comes with psmisc-22.21 and will install it automatically if your psmisc version isn't new enough.
    b. If you want to install XALT to work for several machines then you should probably install psmisc on these machines separately and then make the resulting bin directory part of the path before installing XALT. Then you can install one and only one XALT that should work across machines.
        i. This will likely require installing psmisc a certain way <psmiscpath>/builds/<machine>/bin
        ii. You may need to install xalt once, go through its manual setup phase and then go back and build psmisc per machine AND then edit the xalt ld wrapper in the bin directory so that PSTREE points to the right pstree command per machine by changing the PSTREE setting to use $SYSHOST ASSUMING that xalt_syshost.py returns the generic machine name.
2. cd xalt
3. ./configure --prefix=<xaltpath> [--with-etcDir=ans] [--with-transmission=ans]
    a. --with-etcDir was described above
    b. --with-transmission can be used to pick how you transmit the data collected. By default [file] the data goes in .json files in the users directory ${USER}/.xalt.d/*. This is

recommended at first as its the easiest method to debug. One can also choose directdb or syslog options. These options are described below.

4. make install
    a. The install process places all the code launchers bundled with XALT into the bin directory. You may want to delete the code launchers that are not appropriate for your system so somebody doesn't try to use them by accident. For example, I highly suggest removing the mpirun wrapper from the bin directory on a Cray so that an inexperienced user doesn't think mpirun should work on the Cray.

        cd bin; rm -f ibrun* mpirun srun

5. Local modifications:
    a. site/xalt_syshost.py
        i. We assume a site will want to map names like login1.cheetah, cheetah2 or service2-cheetah to the same name cheetah. This is done in the xalt_syshost.py file. This results in records going into the database all having the same consistent (generic) hostname.
            1. ***However, it is likely some sites may want the actual hostname where the link or job launch was executed to be in the database. If so, this may mean you can't use the SYSHOST variable in ld to generically point at the appropriate pstree command (as mentioned above.)***
        ii. Modify xalt_syshost.py to report the syshost name (darter, titan, ...) for your systems. It will have to work on login and job launcher nodes (for Crays, might require "nidxxxxxx" names.) You can modifiy xalt_syshost.py however you wish, but it needs to only return the desired syshost name.
        iii. This is easy to interactively test with "python xalt_syshost.py".
    b. If an existing wrapper [for mpirun,aprun,ibrun] in bin doesn't exist, then create your own wrapper or modify an existing one.
        i. create the wrapper: usually very simple process by copying an existing wrapper and making a few changes. One should set NTASKS (typically the -np option to mpirun) in the wrapper if it is not available as a runtime environment variable. If not set as environment variable, then we only know what the number of tasks are via the mpirun command line. It is up to xalt_site_pkg.py to use NTASKS or whatever to report the number of tasks to the XALT database.
        ii. create site/xalt_find_exec_YYYY.py in the site directory: copy one of the other similarly named files and make the necessary changes. This file needs to be able to understand any arguments provided to the code launcher that are in the form -option val1 [val2 …]. Any argument that takes one or more values using space as the delimiter have to be specified in this file. Any arguments that don't take values or are specified with -option=val1[:val2] do not need to be specified here.
    c. site/xalt_site_pkg.py may need to be modified by the site to convert system names to our "standard" convention.

6. Create a module for xalt/0.5.0.lua

```
prepend_path{"PATH",<xaltpath>/bin, priority=100}
prepend_path{"PYTHONPATH",<xaltpath>/libexec}
prepend_path{"PYTHONPATH",<xaltpath>/site}          # to get xalt_site_pkg
```

or TCL based:

```
#%Module

proc ModulesHelp { } {
   puts stderr "Sets up users' environment to use XALT."
   puts stderr "XALT is a User Environment Tracking Infrastructure.\
            If you encounter any problems while linking or executing mpirun,\
            please unload this module.
 }

conflict totalview

prepend-path PATH          <xaltpath>/build/bin  1000
prepend-path PYTHONPATH     <xaltpath>/build/libexec
prepend-path PYTHONPATH     <xaltpath>/build/site
# ALT_LINKER makes XALT work with the Cray compiler/linker
setenv  ALT_LINKER     /sw/tools/xalt/build/bin/ld
```

Note that the "prepend-path PATH <xaltpath>/build/bin 1000" statement works with versions of Tcl Modules 3.2.6 and beyond. This statement causes an error with Tcl Modules 3.1.6.

The "1000" is a priority setting in Lmod. If you are not using Lmod, then you can remove it, but as noted above recent Tcl Module versions ignore it.

The following is an example layout of the xalt installation not including source directory. This assumes multiple machines, and installing xalt in the same filesystem. If separate filesystems, then you could remove a level (machine) or two (machine and "builds") and just build it for the one machine only and then replicate the hierarchy on other machines.

```
xalt
   |-- builds
   |   |-- machine1
   |   |   |-- bin
   |   |   |-- libexec
   |   |   |-- psmisc
   |   |   |-- sbin
   |   |   |-- site
   |   |   |-- etc
   |   |   `   `-- reverseMapD
```

```
|   |-- machine2 -> machine3
|   |-- machine3
|   |   |-- bin
|   |   |-- libexec
|   |   |-- psmisc
|   |   |-- sbin
|   |   |-- site
|   |   |-- etc
|   |   `   `-- reverseMapD
|-- modulefiles
|   |-- machine1
|   |-- machine2 -> machine3
`   `-- machine3
```

SETUP next steps

After configuration, the very next thing to do is set up the database.

1. run sbin/conf_create.py to create your own xalt_db.conf file to point to your MySQL server. *This file needs to be put in the XALT_ETC_DIR location that was either specified at configuration(--with-etcDir=ans) or by the environment variable XALT_ETC_DIR.*
2. Use sbin/createDB.py to create the initial db schema.
    a. if the schema created previously, then you may just rerun sbin/conf_create.py to recreate the xalt_db.conf file with appropriate settings, but only if the xalt_db.conf needs to be recreated.

Testing

XALT has multiple ways it can "transmit" the data collected to the database. By default, XALT uses json files written to the user's home directory in the ~/.xalt.d/ directory. However, XALT can be set insert the data into the database directly or it can put the information in syslog.

For now, we assume XALT is set to use the file transmission option via json files.

1. Load the XALT modulefile

    ```
    module use $XALT_DIR/modulefiles/<machine>
    module load xalt
    ```
2. Test the linker hijacking.
    a. module load xalt; CC -o hello hello.c
    b. If this worked there should be a file in the ~/.xalt.d/ called link.darter.....json
3. Submit a job. If this worked then there should be a ~/.xalt.d/run.darter.....json file.
4. Getting data to DB:

a. If you are generating json files: The script sbin/xalt_file_to_db.py reads the json files in ~/.xalt.d and loads them into the database. It will delete files with --delete option.
    i. Where ever you run this command, it needs to find the xalt_db.conf file.
    ii. XALT_USERS: colon separated list of users to find the json file. This can be used to to target test users instead of all users. If not set, then all users (found from getent command) directories will be searched and processed. In most circumstances, this command will need to be run as root.
b. This isn't called automatically anywhere in XALT as we expect this to be called in a cron job by a privileged user to put data into the database.
    i. The wrappers could be modified to call this function, but then the "direct to database" option would be better.

Other Transmission methods:

A very important note is that the transmission method can be set at configuration time (defaults to file), but also can be set at any time later with an environment variable. So a site can first set it up to use json files, but then can switch to syslog or direct to database at any time. This is done with the XALT_TRANSMISSION_STYLE environment variable with options of: file, syslog, directdb.

The file transmission method has a big concern from the developers - what happens in the case of someone running thousands upon thousands of small jobs concurrently on a large machine? This could generate thousands and thousands of json files in the home directory of that user -- will there be enough space? enough inodes? too much NFS traffic in general? If this is not a concern at your site, then the file method should be sufficient

syslog

The syslog method for transmission is very similar to the file method. First the data goes to syslog, and then asynchronously (via a cron job) the data must be collected and put into the database.

We expect this method to be the best for production use. To use syslog (and in this context we mean rsyslog since that is all we have tested), you will need to

- set up a configuration file for syslog and place in /etc/rsyslog.d/ that we called xalt_syslog.conf

  ```
  $MaxMessageSize 256k
  if $programname contains 'XALT_LOGGING' then /var/log/xalt.log
  & ~
  ```

  This example shows that the log file is set up as /var/log/xalt.log. Note that /var is probably local for the node where the linker or job launcher is run. This is fine, but you will have to run the syslog parser on each node for this setup. Alternatively, it might be easier if you had say one node/server where all the log files could be put by syslog and then you would only have to run the parser on that node, but for each file.

Also note that 64k is the maximum message size as given. We have already hit a case where a link line was larger than this and this results in the XALT log message being incomplete and as a result the parser will have to skip those entries as incomplete. We previously used 64k because all link lines (until this case) were much smaller and all our run examples were larger but less than 256k.

- modify /etc/rsyslog.conf to use this new configuration

```
# Include all config files in /etc/rsyslog.d/
$IncludeConfig /etc/rsyslog.d/*.conf
```

- restart rsyslog
- set up rotation on the /var/log/xalt.log log file with a logrotate configuration file like

```
/etc/logrotate.d>  cat xalt
/var/log/xalt.log{
  copytruncate
  rotate 4
  daily
  create 0644 root root
  missingok
}
```

The above sets a 4 day rotation on the files. We suggest nothing less than 2. The above is also setting the log file to be readable by all. This is a site dependent setting - in this case, a non-root account can be used to parse the data and put it in the database.

- use xalt_syslog_to_db.py to collect data from syslog

```
python xalt_syslog_to_db.py /var/log/xalt.log.1
```

We assume the installers know that all of these steps will have to be done on each of the nodes where the linker and the job launcher (mpirun, aprun, etc) will be run.

direct to database

The direct to database method is probably the simplest. It basically inserts the data into the database as it is being collected. This is how the ALTD infrastructure worked for years.

This method has the security concerns of “users modify the database directly”. Yes they do, but only through the wrappers and most if not all users have no idea that is happening. To ameliorate this concern, it is important to use an “insert-only” account for these transactions. There are also concerns about many database transactions going on all the time. Anecdotally, this has never been reported as a problem on several large HPC installations with thousands of users, but it certainly could be depending on how the database is set up.

To use this method, just set XALT_TRANSMISSION_STYLE to directdb. Really nothing else to do.

TIP: It is only in directdb mode that users would ever see any issues accessing the database (whether it be permissions, connectivity, performance.) In the other modes, the data is seemlessly

written and the users never know. If there are any problems, the admins can deal with it asynchronously without users ever seeing any problems.

Installation Considerations

This section describes considerations when installing XALT. Installation is currently a semi-automatic process.   XALT is written in Python, however there are some additional packages described previously that require compilation.

The XALT prototype has been installed on CRAY XE/XCs, an SGI Ultraviolet, and several Intel-based clusters at four different centers: National Institute for Computational Sciences (NICS) at Oak Ridge National Laboratory (ORNL), Texas Advanced Computing Center (TACC), Los Alamos National Laboratory (LANL), and the Swiss National Supercomputing Center (CSCS). The installation process has been fairly straightforward, as intended, and more architectures will be supported in time.

Intercepting linkers and job launchers:

If a site has multiple code launchers or linkers, then the site has to decide how they want to intercept them. If they are already wrapped, then our wrappers could be inlined or called by the existing wrappers. If one is wrapping the linker or code launcher for the first time, there are several options to consider. If the site has multiple launchers, for example, then there must be a module system or other mechanism that allows users to swap the version of launcher. We'll assume this is the case for the rest of this discussion and use mpirun as the example of the parallel code target to be wrapped. There are a variety of options for wrapping. Each will be discussed in turn.

- **Move launcher**: One can physically move each launcher out of the way by renaming it; for example, rename ld and mpirun to ld.x and mpirun.x, respectively and then actually place the XALT ld and mpirun wrappers in /usr/bin or in the MPI installation directories. Then the XALT wrapper can replace of the original mpirun. This is not our suggested option as it requires modifications to our wrapper. If in /usr/bin, then the modulefile for each MPI must set some environment variable that the ld and mpirun wrappers recognize. If, however, the XALT wrappers are placed in each each MPI installation directory (not recommended), then the wrappers just call "./ld.x" and "./mpirun.x". Within the ld and mpirun wrappers, you still need to set the XALT_PATH properly to where-ever it was unpacked and configured.
- **XALT modulefile**: Create an XALT modulefile that when loaded puts the launcher first in the path. An example modulefile is provided with the xalt.tar file. This can be used to help make XALT part of the default environment. The modulefile modifies the default user PATH and puts the ld and mpirun wrappers first in the PATH. This requires addressing an issue: any change to the MPI library by a module swap will not keep the

XALT wrapper first in the path. To address this additional issue, one can take either of the following approaches:

  - Have each MPI modulefile reload the XALT modulefile (or inline the XALT modulefile contents).
  - Use Lmod [9] as the module system which allows one to specify priorities on a PATH setting. Lmod has the ability to prioritize modules to keep them first in the path[1]. Set the XALT PATH priority to ensure that the MPI modulefile swaps will not be a problem.

  Some sites may already have wrapped mpirun (or their job launcher), in this case all that is needed is to modify the existing wrapper to incorporate or call our launcher. If you don't have mpirun wrapped already, we provide several stub mpirun-like scripts that call.

- **Use Aliasing**: Aliasing can be used to get in front of the code launchers. Each alias points to a unique Python script like xalt_alias.py. An analysis of the command line gives the name of the original command along with some interesting parameters (name of the compiler, executable or job file, requested number of processors) and depending on this information the wrapper eventually executes commands to add new information to the database, modifies the running environment, or patches a job file before submitting it. The script runs the same command line, this time in an environment unaware of the aliases.

Env Var BlackList

There is an environment variable blacklist, list of variables discarded, in the xalt_run_submission.py code. You can add or remove from this list, but note that as long as this is part of the xalt_run_submission.py code, code updates will overwrite any changes a site does.

Creating the Reverse Map

The reverse map has been mentioned several times before, but exactly how do you create it. As described above, Lmod can be used as a module replacement. But even if you don't want to replace TCL modules, you can use Lmod to create what we refer to as the reverse map. Basically, it maps libraries (with paths) back to modulefiles.

If you have your modulefiles set up with a one-to-one to mapping of modules to package installations, then Lmod can probably create the reverse map without issue. But on some machines, a module points (with appropriate if tests) can point to a variety of installations and set environment variables depending on the currently loaded compilers and MPI. In this

[1] Another way is to put all the XALT modulefile settings inside each MPI modulefile so that as you swap MPI modules, the XALT wrapper is first in the path.

scenario, the reverse map can be created, but it is a looser reverse map with many-to-one relationships.

And further on some machines (like Crays), for spider to work, you have to run it multiple times (one for each Programming Environment) to get multiple reverse maps, which then have to be combined together for a master reverse map. Below you will see a sample script for how to do this.

The reverse map needs to be created/updated per machine every time a new modulefile or package is installed. So it either has to become part of the software installation process, or run as a cron job every week for example.

And if you have multiple machines and one XALT installation, then you will need to have a reversemap for each machine. That means as an example the etc directory will likely need to have subdirectories for each machine and a reverseMapD in each of those directories. And you will need to set XALT_ETC_DIR in the modulefile for each machine to point to the appropriate place.

Examples

Simple command to create reverseMap
spider -o jsonReverseMapT $LMOD_MODULEPATH > rmapD/jsonReverseMapT.json

Cray script to create reverseMap
An example script, darter_build_rmapT.sh, for a Cray XC30 that uses Lmod (namely the spider utility) to create the reverseMap is provided in the contrib/build_reverseMapT_cray/ directory.

```
#!/bin/bash

########################################################################
# To get this to work please do the following:
#   a) Modify the Site Specific Settings to match your site
#   b) Make sure that this script and the python script
#      "merge_json_files.py" are in the same directory.
#   c) Make sure the module command is defined by using $BASH_ENV
#      or define the module command here.
#   d) Make sure that LMOD_DIR is defined as well
#         (it is defined by $BASH_ENV).
#
########################################################################
```

```
#  Site Specific Setting
########################################################################



BASE_MODULE_PATH=/opt/modulefiles:/opt/cray/ari/modulefiles:/opt/cray/crayp
e/default/modulefiles:/sw/local/modulefiles:/sw/xc30/modulefiles:/sw/xc30/m
odulefiles:/sw/local/modulefiles:/opt/cray/craype/default/modulefiles:/opt/
cray/modulefiles:/opt/modulefiles:/cm/local/modulefiles:/cm/shared/modulefi
les

  ADMIN_DIR=$HOME/XALT/use_xalt
  RmapDir=$ADMIN_DIR/reverseMapD

  PrgEnvA=("PrgEnv-cray"  "PrgEnv-gnu" "PrgEnv-intel")

  moduleA=( "PrgEnv-cray/5.2.25" "PrgEnv-gnu/5.2.25" "PrgEnv-intel/5.2.25")

  ########################################################################
  # must define the module command and $LMOD_DIR:
  ########################################################################

  if [ -f "$BASH_ENV" ]; then
    source $BASH_ENV
  fi

########################################################################
#  End Site Specific Setting
########################################################################

if [ ! -d $RmapDir ]; then
  mkdir -p $RmapDir
fi

SCRIPT_DIR=$(cd $(dirname $(readlink -f "$0")) && pwd)
PATH=$SCRIPT_DIR:$LMOD_DIR:$PATH

cd $RmapDir

module unload "${PrgEnvA[@]}" 2> /dev/null
prev=""
for m in "${moduleA[@]}"; do
    sn=$(dirname $m)
```

```
    v=${m##*/}

    module unload $prev  2> /dev/null
    module load $m      2> /dev/null
    prev=$m

    echo -n '-'
    spider --preload -o jsonReverseMapT $BASE_MODULE_PATH >
rmapT_${sn}_${v}.JSON
    echo -n '*'

done

echo "*"

OLD=$RmapDir/jsonReverseMapT.old.json
NEW=$RmapDir/jsonReverseMapT.new.json
RESULT=$RmapDir/jsonReverseMapT.json

set -x
merge_json_files.py rmapT_*.JSON > $NEW
if [ "$?" = 0 ]; then
  chmod 644 $NEW
  if [ -f $RESULT ]; then
    cp -p $RESULT $OLD
  fi
  mv $NEW $RESULT
fi

rm rmapT_*.JSON
```

Known issues:

An important issue is the interaction of tools like the debugger Totalview with a job launcher: Totalview needs to interact with the actual job launcher rather than a wrapper. A site can either unload XALT when Totalview and other similar tools are loaded, or edit the Totalview wrappers themselves so they interact directly with the renamed job launcher. Either way, the Totalview runs will not show up in the XALT database.

A Cray specific problem is how the Cray compiler wrapper uses a binutils linker that Cray bundles with the Cray compiler - this is fairly easy to deal with by setting the Cray environment variable ALT_LINKER in the XALT modulefile to point to the XALT wrapper.

```
# ALT_LINKER makes XALT work with the Cray compiler/linker
setenv  ALT_LINKER     XALT_DIR/bin/ld
```

Appendix A - Alternative Technologies

Except for the prototypes on which XALT is built (ALTD and Lariat), to the best of our knowledge no tool has been developed for the explicit goal and objectives presented above. There are a few approaches that are related (some more strongly related than others), however they were designed for other purposes and as such are not good solutions for what we have described. In this section we highlight a few of these approaches, including the ALTD and Lariat prototypes that have led to XALT.

XALT predecessor (ALTD and Lariat): As discussed above, library tracking tools like ALTD and Lariat are already implemented at the respective sites and have demonstrated clearly how useful it is to have this infrastructure in place. However having a single infrastructure, which provides the best of both successful predecessors, is highly desirable. ALTD tracked static and shared libraries and executables. Lariat only tracked shared libraries and executables at runtime and checked for runtime issues. XALT incorporates all the above and adds several more features.

Use of signature matching tools: Signature matching is a common strategy employed by anti-virus software to search for known patterns of data within the program binaries. Efforts [21] had been made to extract information (such as compilers and libraries used) using open-source anti-virus package ClamAV [22]. It comprises of two tools: a signature generator and a signature scanner. The signature generator takes ELF files and automatically outputs ClamAV-formatted signatures files. The signature scanner takes as input the signature files and the executable binaries and outputs all possible match. However, this approach has a few drawbacks: (1) the ELF file (.comment section) is not a guaranteed source of compiler provenance information, (2) if different source files are compiled with different compilers, the resulting program binary will likely lack the compiler-specific code snippets one would need, (3) this approach takes a 'noticable' amount of time to extract signatures (e.g., 28 seconds to scan through a library of 210 MB size [21]), and (4) one has to be vigiliant to make sure that all the signatures are up-to-date.

And the fingerprint work by Luca Clementi; the Fingerprint source is at https://github.com/rocksclusters/FingerPrint. FingerPrint is a software tool which can analyze arbitrary lists of binaries and save all their dependencies information in a file (called Swirl) along with other information.

To capture the libraries used inside a program is not a trivial task, since not all the libraries require calling initialization function or adding the header file in the source code. In fact for example, programs using MPI library (Gropp, Lusk, & Skjellum, 1999) or PETSc library (Balay, et al., 2008) impose an initialization function (MPI_INIT and PetscInitialize), while LAPACK (Anderson, et al., 1999) or NetCFD (Rew, 1990) do not have such functions.

State of the art profiling and tracing tools such as CrayPAT (Cray), Vampir (Solchenbach, 1996), and TAU (Malony, 2006) perform analysis for only one user and provide all the functions call in the application. These tools could provide similar information as a by-product, but they are heavy-weight and introduce compile-time and runtime overheads that should not affect all users all the time.

In the same scope, IPM (Integrated Performance Monitoring) (J. Borrill, 2005) provides a performance profile on a batch job maintains low overhead by using a unique hashing approach that allows a fixed memory footprint and minimal CPU usage. However, XALT is not intended to track all the code execution but rather the libraries used at link time and at the execution. This avoids any overhead when the user executes his code.

Some commercial or open source distributed resources manager such as TORQUE (Staples, 2006) installed on Kraken, help the administrators to monitor supercomputing systems, however they can only observe the number of time the softwares have been used or the CPU hours. This method of extraction of data is called process accounting which records accounting information for the system resources used, their allocation among users. In Linux, thanks to environments commands such as lastcomm (Tam, 2001), which prints out previously executed commands; administrators can parse the output to retrieve summaries on softwares usage. Nevertheless, both TORQUE and process accounting commands reports only the applications called in a batch environment and from job scripts respectively; which results in not taking into accounts libraries or applications called inside a program or a script while XALT does it.

We can mention the work on TOPAS (Mohr, 1999), a tool that monitors automatically the usage and performance on the CRAY T3E by modifying the UNISCOS/mk compiler wrapper script, however, it provides only the performances on the machine from all the user on the compiler and the MPI library used. We can also cite, in PAPI, there is a function PAPI_get_shared_lib_info that returns a pointer to a structure containing information about the shared library used by the program written only in C (Papi).

Use of prelink and other programs: A software tool FingerPrint [24] can analyze arbitrary lists of libraries and save all their dependency information in a file along with other information that can be used to understand if a given application can run on another system or if some dependencies have been modified since file creation. FingerPrint uses prelink (to remove pre-linking information from libraries and get their hash) and dpkg or rpm (to record package

version and info regarding dependencies). Occasionally prelinking can cause issues with application checkpoint and restart libraries like blcr [25] or other libraries that uses blcr internally. In addition, for dynamic tracing FingerPrint uses POSIX ptrace system call which is hard to implement correctly and consistently.

LD_PRELOAD a dynamic linker is the part of an operating system (OS) that loads and links the shared libraries for an executable when it is executed. The specific operating system and executable format determine how the dynamic linker functions and how it is implemented. Linking is often referred to as a process that is performed at compile time of the executable while a dynamic linker is in actuality a special loader that loads external shared libraries into a running process and then binds those shared libraries dynamically to the running process. The specifics of how a dynamic linker functions is operating system dependent.

The GNU/Linux based operating systems implement a dynamic linker model where a portion of the executable includes a very simple linker stub which causes the operating system to load an external library into memory. This linker stub is added at compile time for the target executable. The linker stub's purpose is to load the real dynamic linker machine code into memory and start the dynamic linker process by executing that newly loaded dynamic linker machine code. While the design of the operating system is to have the executable load the dynamic linker before the target executable's main function is started it however is implemented differently. The operating system knows the location of the dynamic linker and in turn loads that in memory during the process creation. Once the executable is loaded into memory the dynamic linker is already there and linker stub simply executes that code. The reason for this change is due to the fact that ELF binary format was designed for multiple Unix-like operating systems and not just the GNU/Linux operating system. On the GNU/Linux operating system dynamic loaded shared libraries can be identified by the filename's suffix ".so".

References

[1] https://www.xsede.org/news/-/news/item/6159

[2] M. Fahey, B. Hadri, T. Robinson, W. Renaud., “Software Usage on Cray System across Three Centers (NICS, ORNL, CSCS)”, Cray Users Group 2012, Stuttgart, Germany.

[3] NSF Advisory Committee for CyberInfrastructure Task Force on Software for Science and Engineering, http://www.nsf.gov/cise/aci/taskforces/TaskForceReport_Software.pdf

[4] Reproducible Research: Addressing the need for Data and Code sharing in Computational Science. By the Yale Law School Roundtable, http://web.stanford.edu/~vcs/Conferences/RoundtableNov212009/RoundtableOutputDeclaration.pdf

[5] M. Fahey, N. Jones, B. Hadri, B Hitchcock, Automatic Library Tracking Database, http://www.nersc.gov/assets/altdmanual.pdf
[6] https://www.tacc.utexas.edu/tacc-projects/Lmod
[7] Automatic Library Tracking Database Infrastructure, http://www.nersc.gov/users/software/programming-libraries/altd/
[8] http://linux.about.com/library/cmd/blcmdl1_sha1sum.htm
[9] http://sourceforge.net/projects/lmod/
[10] https://www.tacc.utexas.edu/stampede/
[11] Bill Barth, R. Todd Evans, John tacc stats Repository, 2013, https://github.com/billbarth/tacc_stats.
[12] S. Balay et al., "PETSc Users Manual", 2008
[13] E. Anderson et al., LAPACK User's Guide (Third ed.).:SIAM, 1999
[14] R. Rew and G. Davis (1990), NetCDF: an interface for scientific data access, IEEE Comput. Graph. Appl., 10(4), 76–82, doi:10.1109/38.56302.
[15] Cray. Using Cray Performance Analysis Tools, http://docs.cray.com/books/S-2376-41/S-2376-41.pdf
[16] Wolfgang Nagel. Vampir – Performance Optimization, https://www.vampir.eu/
[17] Sameer Shende and Alan Malony, "TAU: The TAU Parallel Performance System," International Journal of High Performance Computing Applications, vol. 20, no.2, pp. 287-311, 2006
[18] J. Carter, L. Oliker, D. Skinner, R. Biswas, J. Borrill, "Integrated Performance Monitoring of a Cosmology Application on Leading HEC Platforms," in International Conference on Parallel Processing: ICPP, 2005
[19] TORQUE-TORQUE resource manager.8, Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, FL, 2006
[20] Albert Tam, Enabling Process Accounting on Linux HOWTO, http://www.tldp.org/HOWTO/text/Process-Accounting
[21] C.-D. Lu, M. D. Jones, T. R. Furlani, "Automatically Mining Program Build Information via Signature Matching," in TeraGrid 11 Workshop, Salt Lake City, UT, 2011.
[22] T. Kojm. http://www.clamav.net.
[23] Bernd Mohr, TOPAS – Automatic Performace Statistics Collection on the CRAY T3E (1999)
[24] https://github.com/rocksclusters/FingerPrint
[25] http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/BLCR/
[26] "Joint Institute for Computational Sciences". http://www.jics.utk.edu.