The Automatic Library Tracking Database

Mark Fahey, Nick Jones, and Bilel Hadri
National Institute for Computational Sciences

ABSTRACT: *A library tracking database has been developed and put into production at the National Institute for Computational Sciences and the Oak Ridge Leadership Computing Facility (both located at Oak Ridge National Laboratory.) The purpose of the library tracking database is to track which libraries are used at link time on Cray XT5 Supercomputers. The database stores the libraries used at link time and also records the executables run in a batch job. With this data, many operationally important questions can be answered such as which libraries are most frequently used and which users are using deprecated libraries or applications. The infrastructure design and reporting mechanisms are presented along with collected production data.*

KEYWORDS: library tracking, Cray XT, ORNL, NICS, OLCF

1. Introduction

The Automatic Library Tracking Database (ALTD) tracks linkage and execution information for applications that are compiled and executed on High Performance Computing (HPC) systems. Supercomputing centers like the National Institute for Computational Science (NICS) and the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory (ORNL) maintain a collection of program libraries and software packages in support of HPC activities across diverse scientific disciplines, and it behooves these centers to know which and how many users utilize these libraries and applications. The role of application support staff at such centers is not limited to the installation of third party packages. The staff must also decide when to deprecate software and remove it from support. For example, staff supporting the two Cray XT5s located at NICS and the OLCF, Kraken and JaguarPF, are responsible for more than a hundred software packages and libraries, each with multiple versions. Over time, support staff will need to change defaults and remove older versions. Without a database like ALTD, the application support staff has to make these decisions based on surveys or personal knowledge. However, decisions utilizing these methods are based on incomplete data, forcing staff to be conservative when deprecating and/or changing default software versions. The accurate data provide by ALTD allows the staff to be much more aggressive when managing supported software. Furthermore, national agencies such as the Department of Energy (DOE) and the National Science Foundation (NSF) may request reports on library and application usage, especially for those libraries that they have funded development. The data from ALTD enables quick accurate replies.

ALTD transparently tracks library and application usage by all users. The framework tracks the libraries linked into an application at compilation time and also tracks executables when they are launched in a batch job. Information from both link time and job launch is gathered into a database, which can then be mined to provide reports. For example, ALTD can generate data on the most or least used library with valuable details such as the version number. This database will help application support staff in their decision process to upgrade, deprecate, or remove libraries. It will also provide the ability to identify users that are still linking against deprecated libraries or using libraries or compilers that are determined to have bugs. Tracking the usage of software not only allows for better quality user support; it makes support more efficient, as well.

This paper is organized as follows. Section 2 highlights the requirements and the design of the project. Section 3 describes the methodology and implementation. Section 4 presents some results

from early data mining efforts, including the most used libraries. Section 5 summarizes the project and its future.

2. Requirements and Design

For the initial release of this project, the Cray XT architecture is the target machine for tracking library usage. In brief, there are wrappers that intercept both the GNU linker (ld) to get the linkage information and the job launcher (aprun) when the code has been executed. Subsequent releases will include support for more job launchers (mpirun, mpiexec, ibrun, …) and support additional HPC architectures. Wrapping the linker and the job launcher through scripts is a simple and efficient way to intercept the information from the users automatically and transparently. Nearly every user will compile a code (thus invoking ld) and will submit a job to the compute nodes (thus invoking aprun.)

ALTD only tracks libraries linked into the applications and does not track function calls. Tracking function calls could easily be done using profiling technologies, if that was desired. However, tracking function calls comes at the cost of significantly increased compile time and application runtime. Furthermore, tracking all function calls does not necessarily increase the understanding of library usage. There would be a huge amount of data to store and most of it would be nearly useless[1]. Therefore, tracking function calls is not desired.

A primary design goal was to minimize any increase in compile time or run time, if at all possible. So a lightweight (almost overhead free) solution that only tracks library usage was implemented. The implementation is described in the next section. Please see Appendix A for a more detailed discussion of alternative technologies.

Requirements

The ALTD design requirements are summarized in the following:

- *Do not change user experience if at all possible:* This requirement was the overriding philosophy while implementing the infrastructure. Since ALTD intercepts the linker and the job launcher, the linker and job launcher wrappers are literally touched by every user. Therefore, the goal was that no matter what the ALTD wrappers did (work or fail), it must not change the user experience. It should be noted that ALTD actually links in its own object file into the user executable and that alteration of the link line can in rare cases change the user experience.
- *Lightweight solution (goal of no overhead):* As mentioned above, the ALTD solution has very little overhead (some at link time), negligible overhead at job launch, and nothing during runtime.
- *Must support statically built executables:* The development environment and target architecture for ALTD officially only supports statically linked executables. ALTD only tracks libraries linked during the linking process, and dynamic libraries that are loaded during the execution phase are not supported.

Key Assumptions

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

- *Only one linker and job launcher to intercept:* This assumption means there are only two binaries to intercept. If a site has more linkers or job launchers, then wrappers for each might need to be provided if they have different names. If they have the same names and just reside in different locations, then one wrapper for each may still suffice.
- *Only want to track libraries (not function calls):* The reasons were described above. If function tracking is desired, then this package is not the solution.
- *Want only libraries actually linked into the application, not everything on original link line:* It is often the case that more libraries are provided on the link line than are actually linked into the application (as is definitely the case on Cray XT systems). ALTD makes sure to only store those libraries that are actually linked into the executable and nothing more.
- *Want libraries and versions if possible:* Version information is not a direct result of ALTD, but rather how libraries are installed and then made available to users. For example, NICS and OLCF use modulefiles to provide environment variables with paths to libraries and applications

[1] It is our contention that centers would at most be interested in tracking the primary driver routines from well-used libraries, and not any auxiliary routines or user-written routines.

that then appear on the link line, which ALTD stores in the database.

- *Trust the system hostname:* We assume that the hostname where the executable is linked or run will correspond to one of the machine tables in the database. If this is not the case (like on the external login nodes for JaguarPF at OLCF), then the ld and/or aprun wrapper must be modified to work with a "target" hostname that will match one or more of the machine tables in the database.

3. Implementation

Linker

Our custom wrapper for the linker (ld) intercepts the user link line. Because more libraries are included on the link line than are actually used, we go through a two-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 users code. Thus, at a high level, two main steps are done:

1. The ld wrapper first generates a record in the **tags** table (see example in Figure 2.b) with an auto-incremented **tag_id** during this step. Once completed, the record will have the **username**, which is retrieved from an environment variable, and the foreign key, **linkline_id**, that is set to a default value (0) along with **exit_code** set to -1; these two fields will be updated in the second phase of the ld script. As shown in the Figure 2.b, if there is a failure in the compilation process, the table is updated with **exit_code** set to -1 and **linkline_id** set to 0. Moreover, in case that the compilation line is the same as the previous one, **linkline_id** is not incremented and it will refer to the **linkline_id** of a previous command. This is the case for the **tag_id** 91132, where user "user1" performed the same linking process consecutively.

   During this phase, assembly code is generated, compiled and stored in the section header of the user's executable. The assembly code contains four fields – ALTD version (Version), build machine (Machine), tag id (Tag_id), the year (Year). The build machine and tag id are two pieces of information that are necessary to be able to accurately track the executable in the jobs table back to the correct machine **link_tags** table. The assembly code (see Figure 1) is surrounded by some identifying text that enables us to find and retrieve this data swiftly in later steps when needed.

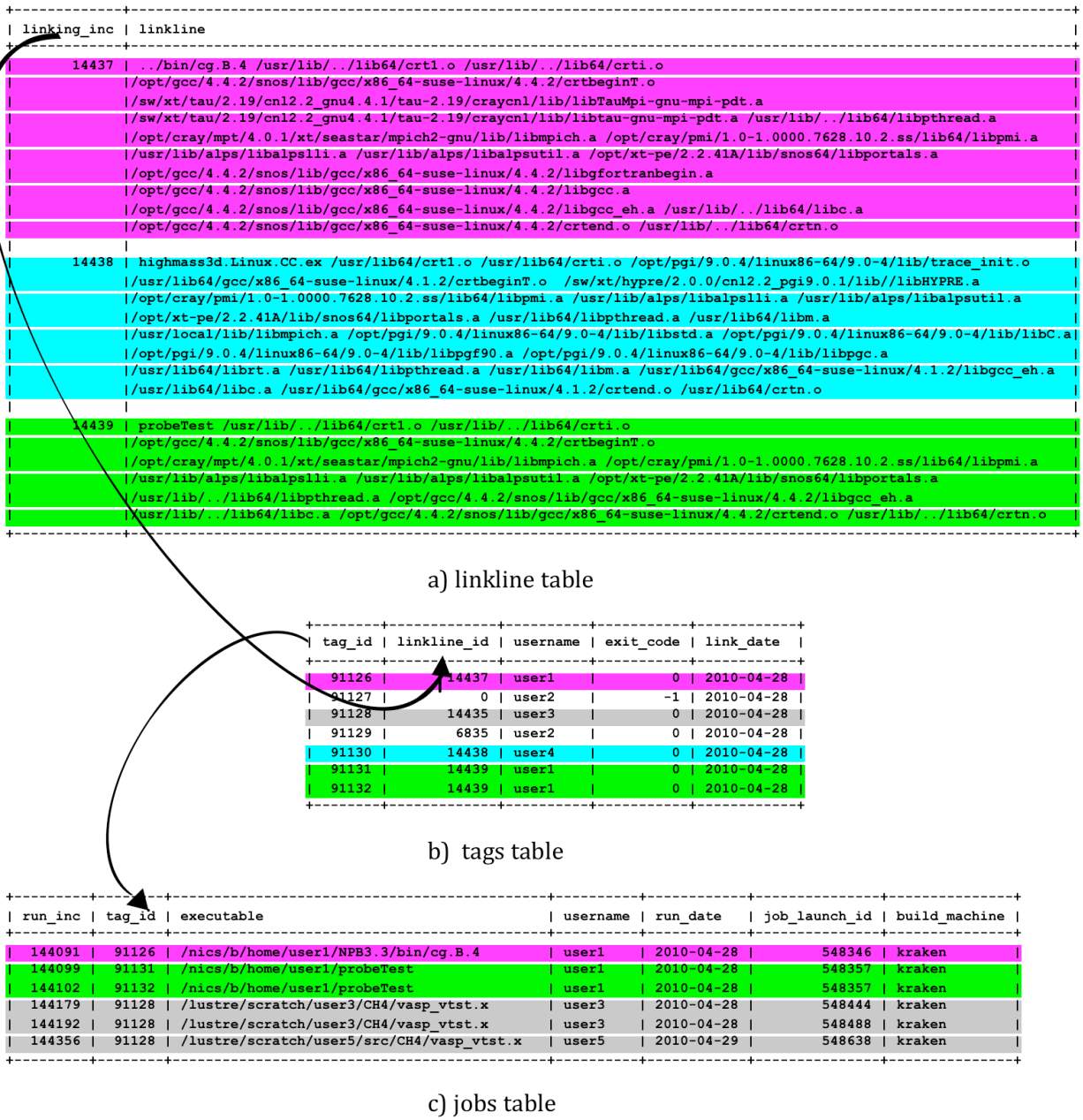
```
.section .altd
.asciz "ALTD_Link_Info"

.byte 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
.asciz "Version:0.7:"
.asciz "Machine:athena:"
.asciz "Tag_id:38:"
.asciz "Year:2009:"
.byte 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
.asciz "ALTD_Link_Info_End"
```

Figure 1 ALTD assembly code

   After generating the assembly code, a bash script is called to check the **tag_id** variable. Regardless of the exit status entry, the real linking status will be stored at this point in a variable for later use. If the tag is 0, then no insert is made, and ALTD exits gracefully. If the **tag_id** is positive, then all the files will be compiled, and the link with "ld –t" (tracemap) is performed. This tracemap is inserted into a temporary file. At this point the temporary file is stripped of unwanted data by sending that file through a few sed rules that remove all .o's that follow a library name, duplicate libraries, ldargs.o, and any random.o created by the compiler, where the random object file may look like /tmp/axj158.o. (If random object files created by the compiler were not removed, then each and every link line would be unique.) The subsequent object code is formed, the object file is added to the link line, and the user's program is linked.

2. In the second phase, the same script used in step 1 is called again to insert or update the **linkline** table. To do so, we use the link line as a search key in the **linkline** table. If the search returns any **linkline_id**, it means the link line already exists; and, if no match is found, then a new link line is inserted and the **linking_inc** index is retrieved. Either way, once you have retrieved the **linking_inc**, the script uses it to update the **linkline_id** in the **tags** table. Finally, the **exit_code** in **tags** table is updated with the status code store, which the bash script stored earlier, and all temp files created by ALTD are removed before exiting with **exit_code**. Figure 2 shows the **linkline**, **tags** and **jobs** tables obtained when loading the MySQL ALTD database.



linking_inc	linkline
14437	../bin/cg.B.4 /usr/lib/../lib64/crt1.o /usr/lib/../lib64/crti.o /opt/gcc/4.4.2/snos/lib/gcc/x86_64-suse-linux/4.4.2/crtbeginT.o /sw/xt/tau/2.19/cnl2.2_gnu4.4.1/tau-2.19/craycnl/lib/libTauMpi-gnu-mpi-pdt.a /sw/xt/tau/2.19/cnl2.2_gnu4.4.1/tau-2.19/craycnl/lib/libtau-gnu-mpi-pdt.a /usr/lib/../lib64/libpthread.a /opt/cray/mpt/4.0.1/xt/seastar/mpich2-gnu/lib/libmpich.a /opt/cray/pmi/1.0-1.0000.7628.10.2.ss/lib64/libpmi.a /usr/lib/alps/libalpslli.a /usr/lib/alps/libalpsutil.a /opt/xt-pe/2.2.41A/lib/snos64/libportals.a /opt/gcc/4.4.2/snos/lib/gcc/x86_64-suse-linux/4.4.2/libgfortranbegin.a /opt/gcc/4.4.2/snos/lib/gcc/x86_64-suse-linux/4.4.2/libgcc.a /opt/gcc/4.4.2/snos/lib/gcc/x86_64-suse-linux/4.4.2/libgcc_eh.a /usr/lib/../lib64/libc.a /opt/gcc/4.4.2/snos/lib/gcc/x86_64-suse-linux/4.4.2/crtend.o /usr/lib/../lib64/crtn.o
14438	highmass3d.Linux.CC.ex /usr/lib64/crt1.o /usr/lib64/crti.o /opt/pgi/9.0.4/linux86-64/9.0-4/lib/trace_init.o /usr/lib64/gcc/x86_64-suse-linux/4.1.2/crtbeginT.o /sw/xt/hypre/2.0.0/cnl2.2_pgi9.0.1/lib//libHYPRE.a /opt/cray/pmi/1.0-1.0000.7628.10.2.ss/lib64/libpmi.a /usr/lib/alps/libalpslli.a /usr/lib/alps/libalpsutil.a /opt/xt-pe/2.2.41A/lib/snos64/libportals.a /usr/lib64/libpthread.a /usr/lib64/libm.a /usr/local/lib/libmpich.a /opt/pgi/9.0.4/linux86-64/9.0-4/lib/libstd.a /opt/pgi/9.0.4/linux86-64/9.0-4/lib/libC.a /opt/pgi/9.0.4/linux86-64/9.0-4/lib/libpgf90.a /opt/pgi/9.0.4/linux86-64/9.0-4/lib/libpgc.a /usr/lib64/librt.a /usr/lib64/libpthread.a /usr/lib64/libm.a /usr/lib64/gcc/x86_64-suse-linux/4.1.2/libgcc_eh.a /usr/lib64/libc.a /usr/lib64/gcc/x86_64-suse-linux/4.1.2/crtend.o /usr/lib64/crtn.o
14439	probeTest /usr/lib/../lib64/crt1.o /usr/lib/../lib64/crti.o /opt/gcc/4.4.2/snos/lib/gcc/x86_64-suse-linux/4.4.2/crtbeginT.o /opt/cray/mpt/4.0.1/xt/seastar/mpich2-gnu/lib/libmpich.a /opt/cray/pmi/1.0-1.0000.7628.10.2.ss/lib64/libpmi.a /usr/lib/alps/libalpslli.a /usr/lib/alps/libalpsutil.a /opt/xt-pe/2.2.41A/lib/snos64/libportals.a /usr/lib/../lib64/libpthread.a /opt/gcc/4.4.2/snos/lib/gcc/x86_64-suse-linux/4.4.2/libgcc_eh.a /usr/lib/../lib64/libc.a /opt/gcc/4.4.2/snos/lib/gcc/x86_64-suse-linux/4.4.2/crtend.o /usr/lib/../lib64/crtn.o

a) linkline table

tag_id	linkline_id	username	exit_code	link_date
91126	14437	user1	0	2010-04-28
91127	0	user2	-1	2010-04-28
91128	14435	user3	0	2010-04-28
91129	6835	user2	0	2010-04-28
91130	14438	user4	0	2010-04-28
91131	14439	user1	0	2010-04-28
91132	14439	user1	0	2010-04-28

b) tags table

run_inc	tag_id	executable	username	run_date	job_launch_id	build_machine
144091	91126	/nics/b/home/user1/NPB3.3/bin/cg.B.4	user1	2010-04-28	548346	kraken
144099	91131	/nics/b/home/user1/probeTest	user1	2010-04-28	548357	kraken
144102	91132	/nics/b/home/user1/probeTest	user1	2010-04-28	548357	kraken
144179	91128	/lustre/scratch/user3/CH4/vasp_vtst.x	user3	2010-04-28	548444	kraken
144192	91128	/lustre/scratch/user3/CH4/vasp_vtst.x	user3	2010-04-28	548488	kraken
144356	91128	/lustre/scratch/user5/src/CH4/vasp_vtst.x	user5	2010-04-29	548638	kraken

c) jobs table

Figure 2 ALTD database tables: a) linkline table, b) tags table and c) jobs table

Notes

There are three types of program libraries: static, shared, and dynamic [1]. For most executables built on a Cray XT, ALTD is able to track them because they are static or they use shared libraries in rare cases. However, libraries that are loaded and unloaded at runtime such as dynamically linked libraries are not tracked since ALTD retrieves the information during the link process. Both static and shared libraries called inside a program are linked during the compilation processing and therefore tracked. By wrapping only the linker ld, the first version of ALTD will be able to track the static and shared libraries when it is added in the link process, and the tracking of dynamic libraries will be considered for future development.

Job Launcher

Launching parallel jobs on compute nodes is typically done through a parallel job launcher such as mpirun, mpiexec, or aprun and often within a batch system (like PBS or LoadLeveler). Interactive support for parallel jobs is often limited, if even available. The job launcher is intercepted as a

secondary measure of "library usage" by counting how many times an executable is run and thus in turn how many times the libraries are used by linking the jobs table data back to the **linkline** table. This is different than counting the number of times a library was used in a link line.

On the Cray XT5 systems, the job launcher is "aprun" used within a PBS (Torque) job to run compiled applications across one or more compute nodes. In the following, the description of the job launcher interception is discussed. (The method is portable, but the current implementation is designed for aprun and as such would require modifications to work on other architectures.) On both Kraken and JaguarPF, the aprun job launcher was already "wrapped" before ALTD was deployed. To work with the wrappers that were already in place, an "aprun-prologue" script was implemented. The aprun wrapper calls this script to do the ALTD job tracking.

1. The aprun-prologue extracts information on PBS environment variables such as the working directory (PBS_O_WORKDIR) and the job id (PBS_JOBID).
2. The command objdump is run on the executable to extract the information that has been stored in the ALTD section header of the user's executable during the link process.
3. The extracted information is then inserted in the **jobs** table, and then control is passed back to the aprun wrapper that eventually calls the real aprun. Figure 2.c shows examples of final output in the **jobs** table.

Production

To integrate the wrappers into production, we suggest adding the wrappers to the default environment. This can be done in a variety of ways. The following discusses two possibilities, with the first being recommended.

Using a modulefile

A modulefile can be used to make ALTD part of the default environment. The modulefile modifies the default user PATH and puts the ld and aprun (or aprun-prologue) wrappers first in the PATH. One must make sure that the modulefile has a variety of ALTD-related environment variables set appropriately. This method gives the user the ability to unload the ALTD module if it somehow causes a problem.

The only known problem is the interaction of tools like Totalview [2] with the job launcher – Totalview needs to interact with the real job launcher not a wrapper. A site can either unload ALTD when Totalview and other similar tools are loaded or they can edit the Totalview wrappers themselves so they interact directly with the real job launcher. On Kraken, instead of additionally wrapping Totalview, the Totalview modulefile was modified to automatically unload the ALTD modulefile. This has the drawback that aprun invocations via Totalview are not tracked.

This method has the potential benefit of being more scalable. If one has multiple linkers or job launchers with the same name in different locations, loading the ALTD module ensures that the wrappers are then "in front" of the various executables.

Linker and job launcher relocation

Another installation method is to rename ld and aprun to ld.x and aprun.x, respectively before placing the ALTD ld and aprun wrappers in /usr/bin. Within the ld and aprun wrappers, the wrappers still need to be configured to point to the location where the ALTD configuration files reside.

As with the modulefile method, a known problem is the interaction of tools like Totalview with the job launcher. In this scenario, the only solution is to edit the Totalview wrappers themselves so they interact directly with the renamed job launcher aprun.x. The relocation of the real linker and job launcher is how ALTD is implemented on JaguarPF. Therefore, as indicated above, the Totalview wrapper had to be edited to point to the real aprun.

This method is less scalable and maintainable in the scenario of having multiple linkers or job launchers; each and every binary has to be renamed, and the wrappers put in many locations. And if a user does encounter a problem with ALTD, there is no simple fix (like unloading the ALTD module). Instead the user has to figure out that the real linker and job launcher have been renamed.

Notes

Soon after "turning on ALTD", some executables that are tracked by the job launcher wrapper will not have the ALTD section header. And, thus, will not have a corresponding entry in the **linkline** table because they were compiled before ALTD went into production. This is not viewed as a problem with this release; it is just an unfortunate

side effect that goes away over time. The job launcher wrapper could be edited to ignore executables without an ALTD section header (omitting them from the jobs table) so that everything in the job table had a corresponding link line, if desired.

For more details, we refer to the ALTD manual [3].

4. Sample Reports and Impact

In this section, we show a few results from early data mining from the NICS and OLCF databases. The OLCF database has data dating back to January 2009 (based on an earlier implementation of ALTD), and the NICS database has data dating to February 2010. For the latter, making conclusions with only a couple of months of data is clearly premature.

One of the more interesting reports to run at both NICS and OLCF is the most used library. If all libraries are taken into consideration, then the compiler libraries that are associated with the default compiler are always the most used libraries. On both Kraken and JaguarPF, PGI compilers are the default and, thusly, the most used compiler by far; and, the most used version basically corresponds to the particular version that was the default for the longest amount of time. The Cray MPI and portals libraries are the next most often used libraries because every parallel job must link with these libraries. If the compiler and communication (MPI and portals) libraries are filtered out, then the results get more interesting, and some selected results are presented for both NICS (Kraken) and OLCF (JaguarPF) below.

Table 1 and Table 2 show the top 5 libraries most often linked against on both Kraken and JaguarPF, with Table 1 listing software provided by Cray and Table 2 listing software installed by local staff.

Table 1 Top 5 libraries used provided by Cray.

Rank	Kraken	JaguarPF
1	CrayPAT/5.0	CrayPAT/4.x
2	Libsci/10.4	PETSc/3.0
3	PETSc/3.0	PAPI/3.6
4	FFTW/3.2	ACML/4.2[2]
5	HDF5/1.8	HDF5/1.8

[2] Cray's libsci is actually used more than ACML, but the usage is spread out over many versions (due to defaults changing over time.)

Table 2 Top 5 libraries used maintained by center.

Rank	Kraken	JaguarPF
1	SPRNG/2.0b	SZIP/2.1
2	PETSc/2.3	HDF5/1.6
3	Iobuf/beta	Trilinos/9
4	TAU/2.19	PSPLINE/1.0
5	SZIP/2.1	NetCDF/3.6

On Kraken, the most widely used library provided by Cray (residing in /opt) is CrayPAT, Cray's profiling and analysis tools, and the most used third-party library is SPRNG (a parallel random number generator). The most widely used library on JaguarPF is CrayPAT/4.4[3]. The most used third-party library not provided by Cray is SZIP/2.1 (which is most often used in conjunction with HDF5.) Clearly, profiling is widely used on both machines since CrayPAT is the #1 application linked against.

Interestingly, Cray's math library, libsci, does not show up in JaguarPF's Top 5 list in Table 1. A closer look reveals that libsci (over all versions) is actually the 5th most linked against library on JaguarPF. If the versions are dropped in Table 1, then ACML drops out and libsci enters at #5. The high usage of libsci is somewhat expected because it is automatically included on the link line by the Cray compiler wrappers and because the library is comprised of multiple commonly used math libraries like BLAS and LAPACK. However, the actual usage of a particular library, say LAPACK, is masked by its inclusion in libsci (a known limitation for this project when this work started.)

The high usage of SPRNG in Table 2 is entirely by one project doing 3D modeling of jet noise. The tracking database clearly shows many compilations with and without profiling tools, an indication of a development cycle likely doing optimization and/or scalability work.

Table 3 shows the executables that have been run most often on Kraken. Note that this is only tracking those executables that are "launched" via aprun. This table does not take into account number of CPU hours, counting only the absolute number of times

[3] If all HDF5 versions are grouped together, then HDF5 is the most used library on JaguarPF. For this exercise, versions 1.6.x and 1.8.x are considered different libraries.

the executable has been run (since Feb 2010.) The table shows a mix of climate (interpo), molecular dynamics (namd and amber), astrophysics (chimera) and bioinformatics (mpiblast.) Interpo is a pre/post-processing tool used to interpolate from one grid resolution to another for the climate IFS code, and the other top four codes are considered main application codes.

Table 3 Top 5 applications executed on Kraken tracked by ALTD database.

Rank	Library	# instances
1	interpo[4]	60,032
2	namd	8,389
3	amber	5,784
4	chimera	4,000
5	mpiblast	2,917

Another interesting report is that of the least used libraries or applications. With only 3 months of data on Kraken, the usefulness of a least used library report for Kraken is questionable. Therefore, we only report on the least used library on JaguarPF (for the 2009 calendar year); and, the results require interpretation. Many libraries that had zero usage also had utility functions that the users ran instead of linking against the library. Since some of these utilities were built before the tracking started, there was no tracked usage of these binaries. There are other libraries where specific versions had zero usage, which is easily explainable since these versions were never set as the default module version. Nonetheless, a least used report indicates for example that fftpack/5 has never been linked against since tracking began on JaguarPF. As a result, it clearly becomes a candidate for discontinued support if not removal from the software list.

NICS has also been tracking executables in an alternate fashion since Kraken went into production. A MySQL database stores the processed Torque accounting records for every job run on Kraken and the corresponding job scripts. The executables are identified from the job scripts using a set of heuristics that map patterns to application names, and the results are stored in a database. A web interface can then be used to generate metrics on the applications that have been run. These tools do not attempt to track library usage, just executables. All executables can be tracked in this way (with no way to distinguish between user or staff supplied.)

Table 4 shows the top 10 applications based on absolute number of times an executable was found in job scripts by searching for a known list of executable names, for the same date range as Table 3. Table 4 includes executables run by staff while Table 3 does not. This method has a few drawbacks including false positive matches, inability to count executables that appear in loops more than once, and strings that match more than one "application."

Table 4 Top 10 applications executed on Kraken based on Torque job scripts.

Rank	Library	# instances
1	arps[5]	11844
2	amber	6789
3	namd[6]	6450
4	chimera	4473
5	h3d[7]	4270
6	sms	3383
7	sses	3153
8	vasp	3131
9	mpiblast	2919
10	gromacs	2234

Table 3 shows that ALTD ranks interpo, namd, amber, chimera, and mpiblast in the top five. The data pulled from the Torque job scripts in Table 4 does not include interpo, because the heuristic search has not been updated to look for it. Conversely,

[4] The tracking database also shows that interpo was compiled on the NICS XT4 (Athena) and then run on the XT5 (Kraken) 60,032 times.

[5] Arps was only run on the login/service nodes and therefore not tracked by ALTD.

[6] Namd is actually called many more times than reported in Table 4 because some user scripts have the executable inside a loop, and as such only counted once.

[7] The "h3d" string is found in many job scripts, but often is not the executable in those scripts and thus the number reported is much higher than reality.

ALTD did not detect arps because it was run without being launched by aprun.

To be clear, there are fairly simple reasons why the results from ALTD differ from the Torque job scripts. For example, a user can name their directory mdrun for their own project resulting in false positives since mdrun is also the name of a parallel executable associated with gromacs (an molecular dynamics code.) In contrast, ALTD has entries for each time an executable was launched and, therefore, can provide an accurate count for those executables launched by aprun.

5. Conclusions

Retrospective

The ALTD infrastructure was put into production to track library and executable usage while attempting to not change the user experience. In this project, two of the most used commands on the Cray XT architecture were intercepted (ld and aprun.) The largest challenge was deploying this infrastructure that affects everybody without anyone noticing, because any mistake is noticed almost immediately by users. Only a few users have encountered problems with the infrastructure, and all of the problems have been effectively addressed. Furthermore, linking and job launching take negligibly longer than without intercepting, which was a design goal.

Final Remarks

Overall, this project has been a success. The primary goals that were set out at the beginning of the process have been achieved, and we plan to make improvements to the infrastructure in future releases. Ultimately, the system has proven itself already in that we have tens of thousands of records in the databases. The data is being actively mined to determine how best to orient software support in the future.

Acknowledgments

The authors would like to thank our colleagues for their help and input during the deployment of this infrastructure. We would especially like to recognize Blake Hitchcock and Patrick Lu. Blake implemented the first version of the ALTD infrastructure in C, which is still deployed on the Jaguar systems at the OLCF. Patrick re-implemented the ALTD infrastructure in Python for use on the NICS systems.

This research was supported in part by the National Science Foundation. In addition, this research used resources at NICS supported by the National Science Foundation.

This research was also partly sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy under contract number DE-AC05-00OR22725 with UT-Battelle, LLC. This research used resources of the OLCF at ORNL, which is supported by the Office of Science of the U.S. Department of Energy under contract number DE-AC05-00OR22725.

About the Authors

Mark Fahey is the Scientific Computing Group Leader for the National Institute for Computational Sciences at Oak Ridge National Laboratory. He is a long-time CUG member and currently serves as a Director at Large. He can be reached at Oak Ridge National Laboratory, Building 5100, Room 210, P.O. Box 2008 MS6173, Oak Ridge, TN 37831-6173, E-Mail: mfahey@utk.edu.

Nick Jones is a high performance computing systems administrator in the National Institute for Computational Sciences at Oak Ridge National Laboratory. Nick can be reached at Oak Ridge National Laboratory, Building 5100, P.O. Box 2008 MS6173, Oak Ridge, TN 37831-6173, E-Mail: jones@nics.utk.edu.

Bilel Hadri is a Computational Scientist in the National Institute for Computational Sciences at Oak Ridge National Laboratory. He has a PhD in Computer Science and a Masters in Applied Mathematics from the University of Houston. He can be reached at Oak Ridge National Laboratory, Building 5100, Room 218, P.O. Box 2008 MS6173, Oak Ridge, TN, 37831-6173, E-Mail: bhadri@utk.edu.

Works Cited

[1] David Wheeler. Program Library HOWTO. [Online]. http://www.dwheeler.com/program-library/Program-Library-HOWTO.pdf

[2] Totalview Technologies, "Totalview Reference Guide," 2010.

[3] Mark Fahey, Bilel Hadri, and Nick Jones, "ALTD Manaul," National Institute of Computational Sciences, University of Tennessee, Manual in preparation 2010.

[4] Satish Balay et al., "PETSc Users Manual," 2008.

[5] E. Anderson et al., *LAPACK Users' Guide (Third ed.)*.: SIAM, 1999.

[6] Cray. Using Cray Performance Analysis Tools. [Online]. http://docs.cray.com/books/S-2376-41/

[7] Wolfgang Nagel. Vampir - Performance Optimization. [Online]. http://www.vampir.eu/

[8] 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.

[9] 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.

Appendix A

To the best of our knowledge, no tool has been developed for the explicit goal and objectives presented in Section 2. There are other approaches that can be considered, but all have major drawbacks some directly related to the Cray XT architecture. We briefly discuss a few alternative methods in this section.

The first approach to mention is that of adding logging functionality to existing libraries. Since some libraries provide an "init" function (like PETSc [4]) the init function could be modified to log information into a tracking database. This would be fairly straightforward, however each version of the library would have to be modified similarly over time. This becomes a maintenance issue. Furthermore, the bigger problem is what to do with libraries that don't have an "init" function (like BLAS or LAPACK [5].) To track library usage, one would have to insert code into each and every routine just to know if LAPACK was used at all. This is an untenable solution.

Another approach would be to use profiling technologies as briefly discussed in Section 2. State of the art profiling and tracing tools such as CrayPAT [6], Vampir [7], and TAU [8] perform analysis for only one user and provide all the function calls 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 every user all the time. In the same scope, IPM (Integrated Performance Monitoring [9]) can be used to obtain a performance profile. It can do this while maintaining low overhead by using a unique hashing approach that allows a fixed memory footprint and minimal CPU usage.

Yet another approach is to modify the behavior of the dynamic linker during both program linking and execution. With dynamic linking, any function call an application makes to any shared library can be intercepted. Once intercepted, anything can be in that function, including calling the real function the application originally intended to call. To use library interposition, you need to create a special shared library and set the LD_PRELOAD environment variable. As noted in Section 2 under requirements, the method described in this paper has to support statically linked executables and this clearly relies on dynamic linking.

One other tracking mechanism that could be implemented is to log all module loads (and unloads). In this way, any time a library was loaded via a module, that would imply the library was linked into a user program. However, that implication is not necessarily true, and for example, it is unknown into what executable the library was linked.