Recent enhancements to the Automatic Library Tracking Database infrastructure at the Swiss National Supercomputing Centre

Timothy W. Robinson
Swiss National Supercomputing Centre (CSCS)
Via Trevano 131, 6900 Lugano, Switzerland
Email: robinson@cscs.ch

Neil D. Stringfellow
iVEC
26 Dick Perry Ave, Kensington, WA 6151, Australia
Email: ed@ivec.org

***Abstract*—The Automatic Library Tracking Database (ALTD)—an infrastructure developed previously by staff at the National Institute for Computational Sciences (NICS)—is in production today on Cray XT, XE, XK, and XC30 systems at several Cray sites, including NICS, Oak Ridge National Laboratory, the National Energy Research Scientific Computing Center, and the Swiss National Supercomputing Centre (CSCS). The Automatic Library Tracking Database automatically and transparently stores information about applications running on Cray systems and also records which libraries are linked to those applications, and from these data, support staff at HPC centres can derive a wealth of information about software usage—such as the use or non-use of particular compiler suites or the uptake of numerical libraries and third-party applications—right down to the level of specific version numbers. The tool works by intercepting the GNU linker to gather information on compilers and libraries, and intercepting the job launcher to track the execution of applications at launch time. We have recently extended the ALTD framework deployed at CSCS to record more detailed information on the individual jobs executed on our machines: the job information recorded by the previous incarnation of ALTD was limited to username, executable, (batch) job id, and run date; we have extended the tool to record many additional job characteristics such as begin and end times, requested versus used core counts, number of processing elements and threads per process, and mode of linking (e.g., static or dynamic). In combination with custom post-processing scripts—which map executables to software codes, research domains or research groups—our ALTD implementation now delivers a far more complete picture of system usage, providing not only a list of running applications but also information on the way that these same applications are being run. On a practical level, such information can be used, for example, to guide future hardware and software procurements, or to assess whether or not researchers are using our systems in the manner for which they were provided with resource allocations.**

***Keywords*-ALTD; library usage; application usage; monitoring; job accounting**

I. Introduction

In 2005, the Swiss National Supercomputing Centre (CSCS) installed its first Cray system, an XT3 named Piz Palü, which was the very first of its kind in Europe. The early Cray XT systems ran the Unicos/lc operating system and Catamount lightweight kernel, and recorded and stored a great deal of job accounting information, including the complete job launch command (the "yod" line, cf. "aprun" on current systems), and thus the executable name and all command line arguments. During the period of operation of the Cray XT3 it was possible, therefore, to identify with relative ease not only which codes were being run on the system—and the amount of CPU resource they consumed—but also fine-grained information such as the number of processors/cores used for particular codes. This information was invaluable for generating usage reports and for determining which applications were of greatest importance to the user community. As well as the benefits with respect to usage reporting, the accounting database information allowed application support personnel to identify situations where the machine was being used in a less than efficient manner. For example, it was known that certain applications benefitted greatly from using small memory pages. By examining the job accounting database, application staff could see when these applications were launched without the "−small_pages" flag, and action could be taken to educate the users accordingly.

The mapping of executable names to specific software applications requires a certain degree of knowledge on the part of the user support or application specialists. Users have the freedom to name their executables arbitrarily ("a.out", for instance), but our own experience shows that in the vast majority of cases users will choose sensible, descriptive names for their executables, and in the case of third-party applications, users do not tend to rename the executables from their default filenames ("namd2" for NAMD, "cp2k.popt" and "cp2k.psmp" for CP2K, "cpmd.x" for CPMD, and so on). Thus, we can obtain a very reasonable understanding of application usage on our systems from an examination of the executable name and command line arguments.

With the deployment of the Cray XT5 and the introduction of the Compute Linux Environment operating system, we lost the job accounting capabilities that the systems had under Unicos/lc. In 2011, the Application Level Placement Scheduler (ALPS) was enhanced to some extent such that the job launch commands ("aprun" lines) were now recorded in the system log files, but to link these aprun lines to specific batch jobs and/or users would have required a complex set of

scripts that examined multiple different log files on multiple different nodes.

At the present time, the tools provided by Cray give us very little information on which applications are running on the systems. Because of this, application support specialists cannot easily answer questions such as "how many times has application x been launched on the production system in the last month?", or "is someone using a legacy version of application x, and if so, why?" We also lack information on which compilers are being used to build applications, and which libraries are most utilized. We cannot answer questions such as "how many applications make use of library x?", "which users are using legacy versions of library x?", or "how many users are trying out different compiler suites?"

One method of tracking application or library usage is to track the loading of modules. Obviously, this method is only useful for tracking software that has been provided by Cray or HPC centre staff and made available through the modules framework. A second shortcoming of this methodology is that module loads do not necessarily reflect actual usage. Indeed many users load a number of modules by default in their shell initialization files without actually making use of them. Likewise, other users may be making use of centrally installed software without loading the respective module, by providing the paths to "includes" and "lib" directories explicitly in their Makefiles.

User surveys, on the other hand, can provide at best an incomplete picture of software usage, and at worst a highly distorted view. A fundamental problem with user surveys is that they log opinions, and not behaviour: users often simply do not know which software libraries (especially version numbers) they have linked into their applications.

To address the limitations discussed in the preceding paragraphs, we implemented the Automatic Library Tracking Database (ALTD) framework by Fahey, Jones, and Hadri, as described at the Cray User Group meeting in 2010.[1] This tool tracks transparently and completely all library usage and application execution, and is installed on production systems at several Cray sites including the National Institute for Computational Sciences (NICS), the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL), and the National Energy Research Scientific Computing Center (NERSC). Briefly, ALTD records information from every invocation of the GNU linker and from every instance that the resulting executable is launched on the compute nodes. The tool gathers this information by intercepting—through wrappers—the GNU linker (ld) and the ALPS application launcher (aprun). Because the tool parses the entire link line it can be used to determine ancillary information about the compilation, such as which compiler suite was used to build the application. The tool is extremely lightweight and its presence is transparent to the user: there is negligible overhead at link time and job launch, and none at all during job run time.

Application support specialists at HPC centres can be responsible for maintaining very large suites of applications, libraries and tools, many of these with multiple concurrent versions. With ALTD one can determine information such as which libraries and applications are most frequently used (right down to version numbers), which compilers are being used to build particular applications, and which users are running executables linked to old or deprecated libraries (or worse, libraries known to be buggy). This information can assist user support services in managing their supported application portfolios and forecasting needs for future procurements. One of the most powerful features of ALTD is the fact that it tracks only those libraries that are actually linked into the application, not simply a list of all libraries appearing in a user's link line. This is particularly important for the Cray machines because libraries are more often than not managed automagically through the modules framework: some modules are loaded by default (e.g., Cray's scientific libraries, cray-libsci or xt-libsci) and hence the libraries associated with that module appear in the user's link line regardless of whether or not they are actually used.

A. Technical details

For full details of the ALTD framework please refer to Fahey, Jones and Hadri.[1] Briefly, ALTD is written in Python and stores data collected in a MySQL database in three tables: altd_<machine>_link_tags, altd_<machine>_linkline, and altd_<machine>_jobs.

1) Link time: The replacement for ld is a wrapper that generates a record in the link_tags table with an auto-incremented tag_id. The completed record holds the username, a foreign key (linkline_id) and an exit code. In the case where the link line matches exactly with a previous one, the linkline_id is not incremented. The link line is formed as a string and inserted into the linkline table.

During the linking phase, a piece of assembly code is generated, formed into object code and linked into the user's executable. This code—the ALTD section header—consists of four fields: ALTD version number, build machine name, tag_id, and year. The build machine and tag_id are checked at job launch time to trace the executable back to its entry in the link_tags table.

2) Application execution time: The replacement for aprun is a wrapper that can run an arbitrary prologue and/or epilogue script. The wrapper script executes an "objdump" on the executable to obtain the ALTD section header, and an entry is written in the jobs table in the ALTD database. The fields in this table are shown in Figure 1.

B. Extensions

In this paper we describe extensions we have made to the ALTD framework to improve its application-level accounting capabilities. It is important to understand that

Field	Type	Null	Key	Default	Extra
run_inc	int(11)	NO	PRI	NULL	auto_increment
tag_id	int(11)	NO		NULL	
executable	varchar(1024)	NO		NULL	
username	varchar(64)	NO		NULL	
run_date	date	NO		NULL	
job_launch_id	int(11)	NO		NULL	
build_machine	varchar(64)	NO		NULL	

Figure 1. List of the fields in the table altd_<machine>_jobs and the data format of each field.

Field	Type	Null	Key	Default	Extra
aprun_inc	int(11)	NO	PRI	NULL	auto_inc
run_inc	int(11)	NO		NULL	
job_id	int(11)	NO		NULL	
tag_id	int(11)	NO		NULL	
username	varchar(64)	NO		NULL	
account_or_group	varchar(64)	NO		NULL	
begin_time	datetime	NO		NULL	
end_time	datetime	YES		NULL	
executable	varchar(1024)	NO		NULL	
linking	enum('unknown','static', 'dynamic','script')	NO		NULL	
aprun_line	varchar(1024)	NO		NULL	
num_pes	int(11)	NO		NULL	
depth_per_pe	int(11)	NO		NULL	
used_cores	int(11)	NO		NULL	
claimed_cores	int(11)	NO		NULL	
num_nodes	int(11)	NO		NULL	
pes_per_cu	tinyint(4)	NO		NULL	
exit_code	tinyint(4) unsigned	YES		NULL	
some_env_vars	varchar(1024)	YES		NULL	
app_name	varchar(128)	YES		NULL	
notes	varchar(128)	YES		NULL	

Figure 2. List of the fields in the table altd_<machine>_accounting and the data format of each field.

this accounting refers specifically to applications launched with the aprun command, so it is somewhat different from traditional job-level accounting, which is usually dealt with by the batch system in use. We are interested in accounting of compute node activity, whereas job-level accounting would likely consider time spent in moving data around, pre- and post-processing on service nodes, and so on.

In the previous version of ALTD the amount of data captured at run time was somewhat limited, as can be seen in Figure 1. We have added an additional table, altd_<machine>_job_accounting that contains further details about applications launched on the system. The fields in this table are shown in Figure 2, and are described in turn in the following sections. Five fields are in common with the jobs table: run_inc, tag_id, job_id (job_launch_id), executable, and username. Some of the accounting information is generated from modifications to the aprun-prologue and aprun-epilogue scripts, and some from the batch system in use (SLURM in the case of CSCS).

1) aprun_inc: This field is the primary key for the accounting table.

2) account_or_group: This field has been introduced to record the account or group id used for running the job. This is necessary for accounting in situations where a single username is associated with more than one project allocation.

3) begin_time and end_time: These fields hold the timestamps for the application begin and end times. The begin_time is obtained from the aprun prologue script at the point of inserting the record into the database table. Likewise, the end_time is inserted at the time that the aprun epilogue script is executed (at the completion of the "real" aprun command).

It is envisaged that these two fields would be used to determine the wall time of the aprun instance. Then, in conjunction with the used_cores (see below), one could derive the amount of CPU time consumed in the aprun instance. It is important to note that this wall time might be significantly different to the wall time of the batch job enclosing the aprun command: A user may have more than one aprun line in their batch script, and pre- or post-processing commands run on the service nodes may consume considerable additional time.

4) linking: This field categorizes the application launched by aprun as a statically linked executable, a dynamically linked executable, or a shell script. The result is obtained by issuing a "file" command on the executable passed to aprun. If the file command is unable to classify the application as one of these three types, the field is set to 'unknown."

5) aprun_line: This field holds the entire aprun line with all command line arguments. With this information one can see in full detail how a user has launched his/her application.

6) num_pes: This field takes the number of processing elements from the aprun command line argument "-n."

7) depth_per_pes: This field records the number of CPUs used per processing element. It is taken from the aprun command line argument "-d."

8) used_cores: This field records the total number of cores used by the application. This is the product of the number of processing elements and depth per processing element, with core specialization taken into account where applicable.

9) claimed_cores: This field records the total number of cores claimed in the ALPS reservation, calculated as the product of the number of nodes allocated and the number of cores per node as given by a request to apbasil. By comparing the number of used cores with the number of claimed cores application support staff could be alerted to potential misuse of the system: Consider, for example, a situation where a machine is upgraded from a 12-core to 16-core socket, but the user continues to use their old batch job script and thus makes use of only 12 of the 16 processing elements available per socket.

10) num_nodes: This field records the number of nodes used in the job. It is derived from the number of processing elements and their depth, and how many cores are present per compute node.

11) pes_per_cu: This field is relevant for the XC30 architecture, where one can choose how many CPUs are to be used per compute unit for an ALPS job. This is specified by the user with the "-j" option to aprun. An aprun specifying "-j 0" is a request to use all available CPUs per compute unit.

12) exit_code: This field holds the exit code of the aprun process, as recorded by the aprun-epilogue script.

13) some_env_vars: This field could be used to collect information about environment variables set at job launch time, such as $OMP_NUM_THREADS.

14) app_name: This field is provided for ease of mapping executables to application names. It is envisaged that application support specialists would enter this data based on their own knowledge of the application that a given executable is associated with. It is known that certain executable names correspond to certain software packages: For example, "mdrun" is an executable belonging to the molecular dynamics package GROMACS. In this case the app_name field could hold the application name "GROMACS."

15) notes: This field is reserved for application support specialists to add additional information about the application that is necessary for their own reporting requirements. An example might be the scientific domain that the application belongs to.

II. Results

The Automatic Library Tracking Database has been in production on our Cray XE6 (Rosa) since March 2011, our XK7 (Tödi) since October 2012, and the XC30 (Daint) since April 2013. The improvements to the tool described in the previous section were implemented on Rosa and Tödi in October 2012, and on Daint in April this year.

At last year's Cray User Group conference, we presented an analysis of the most used software applications and libraries on Rosa, Kraken and Jaguar.[2] In the present paper we will not present again a list of application and library usage but will focus instead on presenting some examples of scenarios (hypothetical and/or real) where ALTD can be used to determine how applications are being built and run, and how the tool can flag unusual usage patterns or potential misuse of the system. As will be demonstrated, particular queries of the ALTD database have shown some interesting usage patterns on our systems—ones that might not have been revealed by analysing the batch system-level accounting logs.

A. Compiler usage trends

Although not central to its design, ALTD can be used to determine which compiler was used in compiling an executable, due to the fact that ALTD records the complete paths to libraries on the user's link line. To determine the compiler used to build MPI code, one could thus examine the path to the MPI library: The programming environment provided by Cray installs MPI libraries in directories with names that correspond to the compiler (mpich2-cray, mpich2-pgi, and so on), so we can thus infer the compiler used by searching the ALTD database for these very strings.

At CSCS, the Cray, GCC and Intel programming environments are available on all machines, while the PGI and PathScale compilers are available only on the XE6 and XK7. The PathScale programming environment is supported only to a limited extent by Cray and since the upgrade of Monte Rosa to XE6 on 1 December 2011, CSCS users have been strongly encouraged to use an alternative compiler.

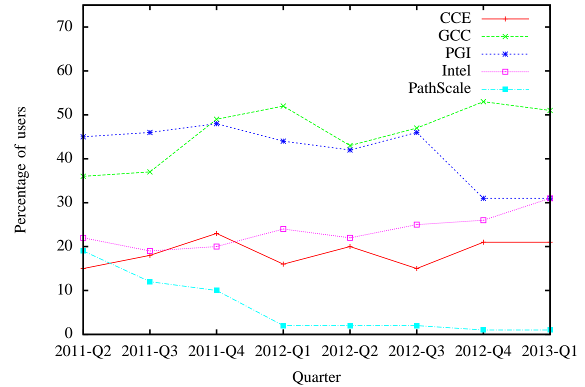

Figure 3. Percentage of users using a particular compiler on the production XE6 system.

Table I
Compiler usage across three systems: percentage of users

	Rosa (XE6)	Tödi (XK7)	Daint (XC30)
Cray	28%	69%	20%
GCC	52%	58%	57%
PGI	44%	19%	N/A
Intel	31%	19%	60%
PathScale	8%	1%	0%

As a case study, we have investigated the use of the various compilers on our main production system over the past two years. In Figure 3 we show how the percentage of users of each compiler has changed over this period of time. At the end of 2011, GCC overtook PGI as the most popular compiler, and it has remained in the top position ever since. In the first quarter of this year, more than 50 percent of all users were using GCC to build MPI code. The use of the Cray compiler has remained more or less constant over time at between 15 and 25 percent of the user base. Uptake of the Intel compiler appears to be steadily increasing, and in the most recent quarter was used by more than 30 percent of the user community. Not surprisingly, with the lack of support for PathScale (the lack of provided scientific libraries, in particular), the use of PathScale has decreased significantly over time: as of today, there are just one or two users continuing to use this compiler on the XE6.

In Table I we compare the overall use of compilers on Rosa to that on our XK7 machine, Tödi, and XC30, Daint. Apart from the use of GCC, which is fairly similar across the three systems (ranging between 50 and 60 percent), the relative use of the compilers differs significantly on the different platforms. On the XK7 the Cray compiler is the

Table II
USERS RUNNING APPLICATIONS HAVING COMPILED OR HAVING NOT COMPILED CODE

	Rosa (XE6)	Tödi (XK7)	Daint (XC30)
Have compiled code	470	153	45
Never compiled code	116	7	9

Table III
LINKNG MODE OF APPLICATIONS RUN ON THREE PLATFORMS

	Rosa (XE6)	Tödi (XK7)	Daint (XC30)
Statically linked	95%	75%	98%
Dynamically linked	5%	22%	2%
Shell script	$< 1\%$	2%	$< 1\%$
Unknown	$< 1\%$	$< 1\%$	$< 1\%$

most popular, being employed by more than two thirds of the user base. It is important to note that the Cray programming environment is loaded by default on the XK7 system: the user can choose a different compiler with a module swap command. For the XE6 and XC30 systems there is no default compiler, and the user must load a programming environment explicitly. This may explain to some extent the high usage of the Cray compiler on the XK7, however there are other factors that may also contribute: Firstly, the XK7 is primarily a research and development system, so the user base may be more willing to try a compiler that they would not have encountered on other platforms. Secondly, the Cray and PGI compilers are the only compilers that currently provide support for OpenACC directives-based programming. The most popular compiler on the XC30 system is Intel, which is perhaps not surprising as Daint is the first Cray system at CSCS to feature Intel-based CPUs.

It is interesting to compare for each platform the number of users compiling code with the number of users running jobs on the system. These statistics can be obtained from ALTD by comparing unique usernames in the jobs and link_tags tables. In Table II we show the number of users who have run jobs on the system without ever having compiled and linked a code. The results show that for the main production system, Rosa, the percentage of active users who have never compiled any code on the system is about 25 percent. These users may be using codes compiled by the HPC centre staff or by their colleagues. This proportion is very similar for the new production system, Daint. For the research and development system, Tödi, on the other hand, we see that less than 5 percent of active users have not compiled code on the system.

The presence (or indeed absence) of “black-box” users is an important consideration because application staff at HPC centres spend a considerable amount of time installing and maintaining third-party applications. In cases where a significant majority of users at an HPC centre are building their own codes, more emphasis should perhaps be placed on assistance porting and optimizing the users’ codes, compared to the amount of time spent installing applications centrally.

B. Linking modes and use of accelerator-based programming models

In Table III we show the number of jobs run on each platform where the application is a statically linked executable, a dynamically linked executable, or a shell script. Ninety-five percent of jobs run on the XE6 were statically linked executables, as were 98 percent of jobs on Daint. The large proportion of statically linked code could be explained by several factors: Firstly, Cray systems have historically supported only static linking, and even today, the compiler wrappers default to static linking. Secondly, static linking might be favoured by users who rely on executables showing completely predictable behaviour over time.

On the GPU-based system Tödi, we see a much higher proportion of dynamically linked executables—about 22 percent of the total. On the XK7 system, loading the cudatoolkit—necessary for CUDA,[1] OpenCL and OpenACC—turns on dynamic linking. Thus we can deduce an upper limit for GPU-accelerated applications at 22 percent of all jobs. Tödi is primarily a machine for developing GPU-accelerated code and these results suggest that there is significant amount of usage on this machine for purposes other than its intention. We can investigate further by looking at the number of users with compile lines that include libcuda.so or libOpenCL.so compared to the total number of users who have compiled any code on this system. The results reveal that 35 of the 153 users who have linked code on Tödi have never built any code using CUDA or OpenCL. These results show that in the absence of system-level GPU accounting—which is still in its infancy—ALTD can provide invaluable information on system usage from simple SQL queries.

C. Job sizes and applications

In Table IV we show the number of jobs of various sizes executed on the three systems. Rosa (in its present incarnation) is a sixteen cabinet XE6 system (1496 compute nodes), Tödi is a three cabinet XK7 (272 compute nodes) and Daint is a twelve cabinet XC30 (2256 compute nodes). The results show that on all but the newest platform the majority of jobs are single node jobs. On Daint there is a large proportion of 62 node jobs, and further querying of the ALTD database shows that these come from a single user. The ALTD accounting table provides further information about the application and the way it is being run: the code is CP2K (cp2k.popt) and was compiled by the user; it is being run in pure MPI mode (no OpenMP) using 16 processes per

[1]CUDA 5 introduced support for separate compilation to produce independent object files, and the ability to combine object files into static libraries.

Table IV
PERCENTAGE OF JOBS OF A GIVEN SIZE ACROSS THREE PLATFORMS

Nodes	Rosa (XE6)	Tödi (XK7)	Daint (XC30)
1	64	79	15
2	11	1	1
$3 - 4$	10	3	4
$5 - 8$	6	1	1
$9 - 16$	5	5	3
$17 - 32$	2	2	6
$33 - 64$	1	2	59
$65 - 128$	1	5	5
$129 - 256$	0.5	2	1
$257 - 512$	< 0.1	< 0.1	2
$513 - 1024$	< 0.1	N/A	1
$1025 - 2048$	< 0.01	N/A	0.5
> 2048	N/A	N/A	1

node and with hyperthreading turned off (pes_per_cu=1)). Likewise, ALTD could be used to determine which applications are being run on single nodes, and by whom.

D. Depth per processing element and OpenMP

An examination of the depth per processing element can give us an indication—to a first approximation—of how many users are running threaded applications. In Table V we show the number of jobs launched where depth per processing element is greater than 1 compared to the total number of jobs launched on each platform. In parentheses we give the corresponding number of unique users running such applications.

The results reveal some interesting differences across the platforms: On the Cray XE6 machine, just four percent of jobs were launched with depth per processing element greater than 1. On the XC30, one third of jobs were launched in this manner, and on the XK7 machine, nearly two thirds. Likewise, just 20 percent of users of Rosa have run any jobs with depth greater than 1, whereas the percentages on Tödi and Daint are 68 and 41, respectively.

Table V
NUMBER OF JOBS RUN USING DEPTH PER PROCESSING ELEMENT GREATER THAN 1 (NUMBER OF DISTINCT USERS IN PARENTHESES)

Nodes	Rosa (XE6)	Tödi (XK7)	Daint (XC30)
All jobs	329054 (325)	164988 (119)	5873 (27)
Depth per PE > 1	13034 (73)	104877 (81)	1929 (11)

III. CONCLUSION

We have described an extension to the Automatic Library Tracking Database that enables more complete information to be recorded about applications launched on Cray systems, information such as the number of processing elements and threads used, the mode of linking, and site-definable metadata such as mappings between executable names and applications or application domains.

We have presented some examples of scenarios where an analysis of results obtained from ALTD can assist application support specialists by alerting them to unusual usage patterns or potential misuse of system resources. Moving forward, we consider that there is a strong need for further development of the tool to provide a framework where the data mining, reporting and alerting is a fully automated process. Ideally, the tool should alert application specialists to situations such as the use of legacy or buggy libraries, or potential wastage of available compute resources.

REFERENCES

[1] M. Fahey, N Jones, and B. Hadri, *The Automatic Library Tracking Database*, Proceedings of the Cray User Group 2010, Edinburgh, United Kingdom.

[2] B. Hadri, M. Fahey, T. Robinson, and W. Renaud, *Software Usage on Cray Systems across Three Centers (NICS, ORNL and CSCS)*, Proceedings of the Cray User Group 2012, Stuttgart, Germany.