# Automatic Library Tracking Database

Mark Fahey

Nick Jones

Bilel Hadri

Blake Hitchcock

Much of this manual appears in the 2010 Cray User Group Proceedings paper with the same title by Mark Fahey.

June 2010

## Table of Contents

## Abstract

The Automatic Library Tracking Database (ALTD) is designed to track linkage and execution information for applications that are compiled and executed, respectively, on High Performance Computing (HPC) systems. This tracking infrastructure has been put into production at the National Institute for Computational Science (NICS) and the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL). The database stores libraries used at link time and also records executables run in batch jobs. With this data, many operationally related questions can be answered, such as, which libraries are most frequently used and which users are using deprecated libraries or applications. For example, this database can be used as a tool to help application support staff decide when to deprecate software and remove them from support. Without such a tool, application support staff often make their decisions based on surveys or personal knowledge, however, these decisions are based on insufficient data and the choices made may not necessarily reflect real usage of the different libraries. Furthermore, national agencies such as DOE and NSF often request reports on library and application usage on HPC systems, especially those libraries they have funded development of.

## 1. Automatic Library Tracking Database

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 and 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.

## 2. 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.

[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.

*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 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. Installation

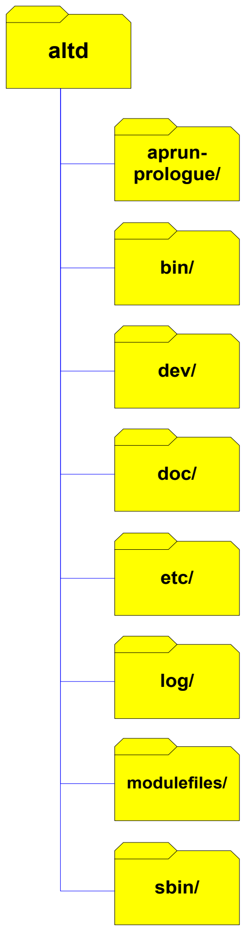
This section describes the installation of the Automatic Library Tracking Database. ALTD is written in Python, so installation does not include any compilation. Installation is a manual process at this time described in below. There are a few prerequisites for running ALTD:

## Prerequisites

- For clients:
  - Python v2.4 or later
  - Python MySQLDb Module
- For servers:
  - MySQL with proper ip ranges opened for client machines

## Unpacking

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



- `<ALTD_DIR>/altd/bin` - contains aprun (possible Cray XT job launcher) wrapper, ld wrapper, pyld.py, and pylauncher. Other job launchers are not available yet. Contributions are welcome.

- `<ALTD_DIR>/aprun-prologue/aprun-prologue` – a prologue script called by the aprun wrapper (most of the work is done here). The aprun prologue script is in its own directory by design – some sites with Cray XTs have aprun wrappers that execute all scripts in a prologue directory and so we did not bundle this in the bin directory.

- `<ALTD_DIR>/altd/etc` - contains configuration files (much more later.)

- `<ALTD_DIR>/altd/log` - contains altd_aprun.log and altd_ld.log

  - Although the altd_aprun.log and altd_ld.log are both stored in `<ALTD_DIR>/altd/log` these locations are customizable, and can be stored in any location including dumped to syslog.

- `<ALTD_DIR>/altd/sbin` - contains table creation script, and python mysql module.

- `<ALTD_DIR>/modulefiles` – example modulefiles to make ATLD part of the default environment

Figure 1: ALTD Directory structure

## Configuring

Note that the current version of ALTD is designed to work on Cray XT systems. We plan to support more systems in the future – any contributions are welcome.

Installation Steps:

1. There are two wrapper scripts that replace the system's default linker and job launcher. With regard to the job launcher, on the Cray XT this is aprun. You will need to modify this script to point to your systems location of python and include the "-E" python option. A sample aprun wrapper is provided, but a site may choose or already have chose to implement their own wrapper and if so doing the custom modifications of the site's wrapper to work with ALTD is completely up to the installer.

So the installer needs to (possibly) modify:

<ALTD_DIR>/bin/pyLD.py

<ALTD_DIR>/bin/aprun  (edit path to real aprun)

<ALTD_DIR>/bin/ld  (edit path to real ld)

The edits to aprun and ld should correspond to how you decide to put the tracking into production (see Production section below.)

And optionally the installer may need to modify aprun-prologue if the site is using an aprun wrapper that implements this prologue.  This file may in turn use other files like a system-hostname file which will need to have the correct system information.

<ALTD_DIR>/aprun-prologue/aprun-prologue

<ALTD_DIR> /aprun-prologue/system-hostname

2. Encode your password in Base64. Take note of the encoded password:

```
# python

Python 2.4.3 (#1, Sep  3 2009, 15:37:37)

[GCC 4.1.2 20080704 (Red Hat 4.1.2-46)] on linux2

Type "help", "copyright", "credits" or "license" for more
information.

>>> import base64

>>> base64.b64encode('xtgl45qz87')

'eHRnbDQ1cXo4Nw=='

CTRL+D
```

3. Now we need to configure the installation.

   Edit <ALTD_DIR>/etc/altd_db.conf  (see figure 2.)

   passwd = Your MySQL password encoded in Base64.

   host = The hostname of your mysql server

   db = The name of the database on the mysql server

   user = The mysql user who will perform the record insertion

4. The next step is to setup the MySQL tables. This install guide assumes that you will have root privileges on the server where MySQL is installed.

   Run MySQL as root on your server:

   ```
   # mysql

   Welcome to the MySQL monitor.  Commands end with ; or \g.

   Your MySQL connection id is 6

   Server version: 5.0.77 Source distribution


   Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
   ```

   A. Create the database. In this example, it is called altd:

      ```
      mysql> create database altd;

      Query OK, 1 row affected (0.00 sec)
      ```

      This should match the name used in altd_db.conf

   B. Create a user to insert into the database. Use the ***non-base64*** version of the password:

      ```
      mysql> create user 'altdinsert'@'%' identified by
      'xtgl45qz87';

      Query OK, 0 rows affected (0.00 sec)


      mysql> grant insert, create, select, update on altd.* to
      'altdinsert'@'%' with grant option;

      Query OK, 0 rows affected (0.00 sec)


      mysql> exit

      Bye
      ```

5. You should now return to the client with the altd tarball installed.

   Change to the <ALTD_DIR>/sbin directory

   Create a symlink from etc/altd_db.conf to sbin/

   ```
   > cd <ALTD_DIR>/sbin
   ```

```
> ln -s ../etc/altd_db.conf
```

6. Now we will create the machine specific tables. Repeat this just *once* for each machine:

```
> python setDb.py

ALTD database configuration file exists!

Do you want to use the file to fill database information?[y/n]y

Machine name:kraken
```

The setDb.py script will log in to the MySQL database and create the necessary tables for you. Repeat this for each machine.

## Configuration Files

Inside the `<ATLD_DIR>/etc` directory are three configurable file; these files enable customization of ALTD and gives administrators the ability to implement the tracking in a way that best suits your environment. Explanations below.

- Edit configuration files in `<ALTD_DIR>/etc`
  - altd.conf (overall settings) See figure 2.
  - altd_db.conf (database access information)
  - altd_log.conf (logging configuration options)

altd.conf – (program settings) *Altd on/off, Selective on per user,* and *Verbosity on/off.*

- enables *setting of user path/ location of ALTD.* (location settings)
- if *Altd_On* and *Select_On* are both set to 1 **ALTD will error.**
- If *Select_On* is set to 1 and *Select_User* is Null **ALTD will error.**

altd_db.conf – (database info file) stores *password, hostname, db name*, and *user name.*

- Make sure proper permissions are set.

altd_log.conf – (logging settings) gives ability to *set log location, set logging level* (verbose level), and *log format.*

- Be careful! ld1/aprun1 log to file only. ld2/aprun2 is enabled by turning on verbosity, and log to both file and console.
- When going into production mode, you should probably turn off all logging unless you are prepared for a lot of data to go into the log directory.

### *Environment variables*

There are six environment variables that are set inside the file etc/altd.conf.

ALTD_ON specify whether the ALTD system is ON or OFF. If ALTD_ON = 1 for every user, unless the user(s) are defined in ALTD_SELECT_OFF_USERS and ALTD_SELECT_ON variable in ignored. We denote If ALTD_ON = 0) for all users, unless the user(s) are defined in ALTD_SELECT_USERS and ALTD_SELECT_OFF_USERS variable is ignored.

```
#!/bin/bash

export ALTD_ON=1
export ALTD_SELECT_ON=0
export ALTD_SELECT_USERS=jones,faheymr
export ALTD_SELECT_OFF_USERS=
export ALTD_VERBOSE=1
export ALTD_PATH=/nics/b/home/jones/altd
export PATH=$ALTD_PATH/bin:$PATH
```

**Figure 2: Example altd.conf configuration file.**

**IMPORTANT**: If the ALTD environment variables are set as follows:

- ALTD_ON and ALTD_SELECT_ON are both set to 1,
- ALTD_ON and ALTD_SELECT_ON are both set to a value different than 1,
- ( ALTD _ON or ALTD_SELECT_ON are both set to 1) and ALTD_PATH is null
- ALTD_SELECT_ON is set to 1 and ALTD_SELECT_USERS is null

ALTD will exit and it will just link without the tracking option.

## Production

At this point, you should be ready to use the ld and aprun wrappers found in <ALTD_DIR>/bin. To integrate these fully, we recommend adding these to the default environment. This can be done in a variety of ways. The following discusses two possibilities with the first being recommended.

### Intercepting with a modulefile

An example modulefile is provided with the altd.tar file. This can be used to help make ATLD part of the default environment. The modulefile modifies the default user PATH and puts the ld and aprun wrappers first in the PATH. One must make sure that the modulefile has ALTD_PATH set appropriately. This method gives the user the ability to unload the ATLD module if it somehow causes a problem.

The only known problem is the interaction of tools like totalview 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 renamed job launcher aprun.x.

Some sites may already have wrapped aprun (or their job launcher), in this case all that is needed to be done is to modify the existing wrapper to call the aprun-prologue. If you don't have aprun wrapped already, we provide a stub aprun that calls our aprun-prologue.

Note that the altd.conf settings can be embedded in the modulefile.

### Intercepting by moving binaries

Another installation method is to, for example, rename ld and aprun to ld.x and aprun.x, respectively and then actually place the ALTD ld and aprun wrappers in /usr/bin. Within the ld and aprun wrappers, you still need to set the ALTD_PATH properly to where-ever it was unpacked and configured.

As above, a known problem is the interaction of tools like totalview with the job launcher – totalview needs to interact with the real job launcher not a wrapper. In this scenario, the only solution is to edit the totalview wrappers themselves so they interact directly with the renamed job launcher aprun.x.

## 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 were not tracked at link time because they were compiled before ALTD went into production. This is not viewed as a problem with this release, just an unfortunate side effect. The job launcher wrapper could be edited to ignore executables without an altd section header (don't add them to the jobs table) so that everything in the job table had a corresponding link line.

## Security concerns

Access to mysql database

Altd_db.conf – readable by users, but users only have insert access to the mysql table

**NICK: Can you add more:**

## License

## 4. Implementation

The objective of ALTD is to track libraries linked into the applications and also to track the executables when launched in a batch job that were linked under the ALTD infrastructure. The ALTD infrastructure is designed to intercept the linker (ld) and the job launcher (aprun on Cray XTs).

### a. Linker (ld)

Figure 3 shows a flowchart how the linker (ld) wrapper intercepts the user link line. To do this, a method was devised to replace the binary ld provided as part of the binutils package with our own scripts. When a compiler (or the user) calls the linker, our script parses the command line to capture the link line.

Here are the main three steps:

1. The ld wrapper checks the ALTD environment variables. If the ALTD environment variables are set correctly, and a workspace is created to hold temporary information, the ld wrapper will call the python file pyLD.py to generate the assembly code and to link the executable and fill out the link_tag table. Otherwise, the wrapper will just link the application without tracking.

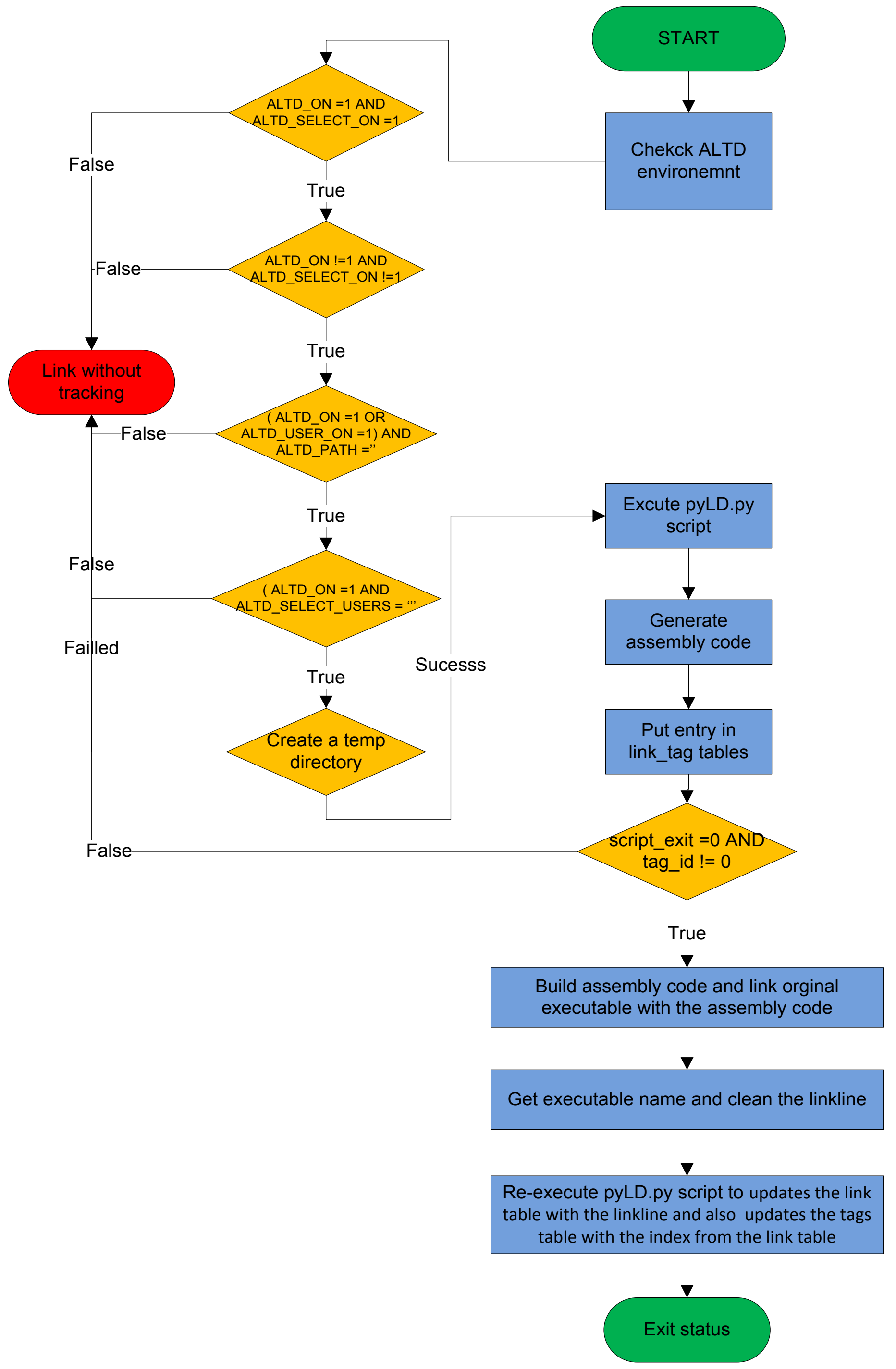
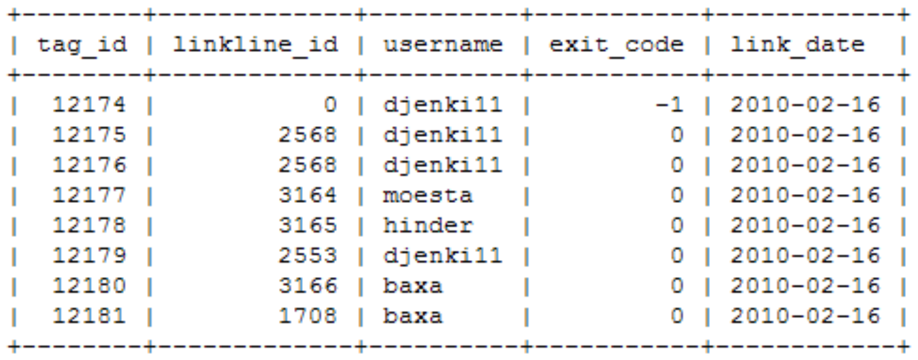

START
Chekck ALTD environemnt
ALTD_ON =1 AND ALTD_SELECT_ON =1
True
False
ALTD_ON !=1 AND ALTD_SELECT_ON !=1
True
False
Link without tracking
( ALTD_ON =1 OR ALTD_USER_ON =1) AND ALTD_PATH =''
True
False
( ALTD_ON =1 AND ALTD_SELECT_USERS = ''
True
False
Failled
Create a temp directory
Sucesss
Excute pyLD.py script
Generate assembly code
Put entry in link_tag tables
script_exit =0 AND tag_id != 0
True
False
Build assembly code and link orginal executable with the assembly code
Get executable name and clean the linkline
Re-execute pyLD.py script to updates the link table with the linkline and also updates the tags table with the index from the link table
Exit status

**Figure 3: Flowchart showing how the ld wrapper intercepts and stores the link line.**

2. Then, the ld script invokes the python file **pyLD.py** where four main functions are implemented (main, table_lookup, print_assembly, update_tag_table).

- The **main** function checks the arguments when ld is called and invokes the functions described below. The main also initialize the MYSQL connection and disconnect it when it is needed to store the data.
- The **table_lookup** function inserts a record to the link_tags table.
- The **print_asssembly** function generates the ALTD assembly code
- The **update_tag_table** function inserts the linkline and updates the records in link_tags table.



```
+--------+-------------+----------+-----------+------------+
| tag_id | linkline_id | username | exit_code | link_date  |
+--------+-------------+----------+-----------+------------+
|  12174 |           0 | djenki11 |        -1 | 2010-02-16 |
|  12175 |        2568 | djenki11 |         0 | 2010-02-16 |
|  12176 |        2568 | djenki11 |         0 | 2010-02-16 |
|  12177 |        3164 | moesta   |         0 | 2010-02-16 |
|  12178 |        3165 | hinder   |         0 | 2010-02-16 |
|  12179 |        2553 | djenki11 |         0 | 2010-02-16 |
|  12180 |        3166 | baxa     |         0 | 2010-02-16 |
|  12181 |        1708 | baxa     |         0 | 2010-02-16 |
+--------+-------------+----------+-----------+------------+
```

Figure 4: Link tags table.

The ld script generates a record in the **link_tags** table (see Figure 4) 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 the exit code set to -1; these two fields will be updated in the second phase of the ld script. As shown in the Figure 4, if there is a failure in the compilation process, the table is filed with the exit code (-1) and the linkline_id is set to 0. Moreover, in case that the compilation line is the same as the previous one, the 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 12176, where the user djenki11 performed the same linking process consecutively.

During this phase, assembly code is generated and it will be compiled and stored in the section header of the user's executable. The assembly code contains four fields – version number of ALTD, 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 is surrounded (see Figure 5) 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 5: ATLD section header assembly code.**

After generating the assembly code, the bash script is called again and checks the tag_id variable. Regardless of the exit status entry, the real linking status will be stored at this point in a variable for use later. If tag is 0 then no insert is made then ALTD exits gracefully. If tag_id is positive, then all the files will be compiled to .o and linking with ld –t (tracemap) is preformed and the map is inserted into a temp file. At this point the temp 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 compile. (If random.o was not removed it would cause the creation of a new linkline everytime.) The subsequent object code is formed, and this object file is added to the linkline and the user's program is linked.

3. In the last step, the python script pyLD.py is called again to insert or update the linkline. To do so, we use the linkline as a search key in the linkline table, if the search returns any id it means the linkline exist, (library has been referenced before) if an id does not exist then we insert a new linkline and retrieve the linking_inc (which is the same as the query from linkline). Either way once you have retrieved the linking_inc the script uses this to update the linkline_id in the tags table. Finally the exit_code in link_tags table is updated with the status code store which the bash script stored earlier, removes all temp files created by altd and exits with exit_code. Figure 6 shows the linkine table obtained when loading the MySQL ALTD database. This linktable corresponds to the link_tags table obtained in the Figure 4.

```
+-------------+-----------------------------------------------------------------------------+
| linking_inc | linkline                                                                    |
+-------------+-----------------------------------------------------------------------------+
|           1 | /usr/lib64/gcc/x86_64-suse-linux/4.1.2/crtbeginT.o /usr/lib64/libm.a       |
|             | /opt/pgi/9.0.3/linux86-64/9.0-3/lib/libpgc.a /usr/lib64/libpthread.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/libgcc_eh.a                          |
|             | /usr/lib64/gcc/x86_64-suse-linux/4.1.2/crtend.o /usr/lib64/crtn.o          |
|             |                                                                             |
|           2 | a.out /usr/lib64/crt1.o /usr/lib64/crti.o                                   |
|             | /opt/pgi/9.0.3/linux86-64/9.0-3/lib/trace_init.o                            |
|             | /usr/lib64/gcc/x86_64-suse-linux/4.1.2/crtbeginT.o                          |
|             | /opt/pgi/9.0.3/linux86-64/9.0-3/lib/libpgc.a /usr/lib64/libpthread.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/libgcc_eh.a                          |
|             | /usr/lib64/gcc/x86_64-suse-linux/4.1.2/crtend.o /usr/lib64/crtn.o          |
|             |                                                                             |
|           3 | torus /usr/lib64/crt1.o /usr/lib64/crti.o                                   |
|             | /opt/pgi/9.0.3/linux86-64/9.0-3/lib/trace_init.o                            |
|             | /usr/lib64/gcc/x86_64-suse-linux/4.1.2/crtbeginT.o torus.o                  |
|             | /usr/lib64/libm.a /opt/pgi/9.0.3/linux86-64/9.0-3/lib/libpgc.a              |
|             | /usr/lib64/libpthread.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/libgcc_eh.a       |
|             | /usr/lib64/gcc/x86_64-suse-linux/4.1.2/crtend.o /usr/lib64/crtn.o          |
|             |                                                                             |
|           4 | torus /usr/lib64/crt1.o /usr/lib64/crti.o                                   |
|             | /opt/pgi/9.0.3/linux86-64/9.0-3/lib/trace_init.o                            |
|             | /usr/lib64/gcc/x86_64-suse-linux/4.1.2/crtbeginT.o                          |
|             | /opt/pgi/9.0.3/linux86-64/9.0-3/lib/libpgc.a /usr/lib64/libpthread.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/libgcc_eh.a                          |
|             | /usr/lib64/gcc/x86_64-suse-linux/4.1.2/crtend.o /usr/lib64/crtn.o          |
+-------------+-----------------------------------------------------------------------------+
```

There are three types of program libraries (static, shared and dynamic) (Wheeler). For most executables built on a Cray XT, ALTD is able to track them because they are static or in rare cases they use shared libraries. However, libraries that are loaded and unloaded at runtime such as the dynamically linked libraries are not tracked since ALTD retrieves the information during the linkage processing. Indeed, both static and shared libraries called inside a program are linked during the compilation processing. For more details on the different types of libraries and the linker, we refer to the Appendix A in the manual. 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 linkage process and we will consider the tracking of dynamic libraries for future development.

### b. Job Launcher

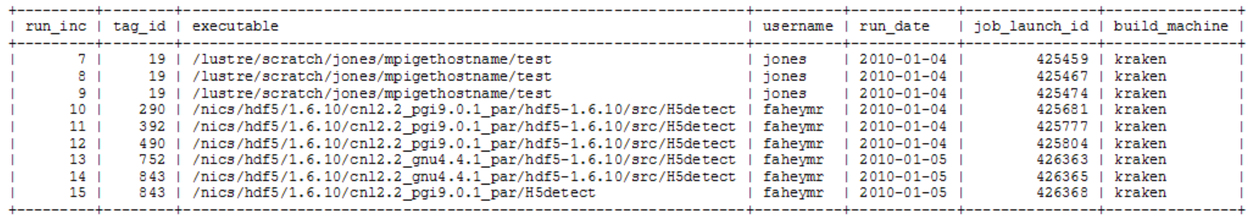
Starting and launching parallel tasks for parallel jobs on assigned hosts are done through an executable script or binary, generally called parallel job launcher such as mpirun, mpiexec or aprun. The job launcher is used to initialize a parallel job from within a PBS batch or interactive environment. It uses the task manager library of PBS to spawn copies of your executable across the resources allocated to your job. It offers also the possibility to control the environment and the execution of an application using numerous parameters, which can either be set by command-line options or by environment variables.

We are interested intercepting the job launcher in order to quantify through the usage of the libraries extracted, through the number of executions.

For instance, on the Cray XT5 systems, to run compiled application program across on one or more compute nodes, the job launcher command is aprun. More information can be found in the man page (Aprun ). In the following, the description of the job launcher interception is done with aprun, however, it is portable for different job launcher and on different architectures.

For ALTD, aprun is a wrapper around the Cray aprun that can run an arbitrary prologue and/or epilogue and it calls the python script aprun-prologue.

1. The aprun-prologue extracts some information on the environment variable such as the directory (PBS_O_WORKDIR) and the job id (PBS_JOBID).
2. Then, the command line of aprun with arguments is parsed to remove all aprun options. ***Is this right?***
3. Then, the command objdump is run on the executable to display the information that has been stored in the section header of the user's executable during the linkage process.
4. The extracted information is then inserted in the MySQL Database, and the aprun is called to complete the process. Figure 7 shows the final output in the MySQL database of the jobs executed.



```
+---------+--------+----------------------------------------------------------------+----------+------------+---------------+---------------+
| run_inc | tag_id | executable                                                     | username | run_date   | job_launch_id | build_machine |
+---------+--------+----------------------------------------------------------------+----------+------------+---------------+---------------+
|       7 |     19 | /lustre/scratch/jones/mpigethostname/test                      | jones    | 2010-01-04 |        425459 | kraken        |
|       8 |     19 | /lustre/scratch/jones/mpigethostname/test                      | jones    | 2010-01-04 |        425467 | kraken        |
|       9 |     19 | /lustre/scratch/jones/mpigethostname/test                      | jones    | 2010-01-04 |        425474 | kraken        |
|      10 |    290 | /nics/hdf5/1.6.10/cnl2.2_pgi9.0.1_par/hdf5-1.6.10/src/H5detect | faheymr  | 2010-01-04 |        425681 | kraken        |
|      11 |    392 | /nics/hdf5/1.6.10/cnl2.2_pgi9.0.1_par/hdf5-1.6.10/src/H5detect | faheymr  | 2010-01-04 |        425777 | kraken        |
|      12 |    490 | /nics/hdf5/1.6.10/cnl2.2_pgi9.0.1_par/hdf5-1.6.10/src/H5detect | faheymr  | 2010-01-04 |        425804 | kraken        |
|      13 |    752 | /nics/hdf5/1.6.10/cnl2.2_gnu4.4.1_par/hdf5-1.6.10/src/H5detect | faheymr  | 2010-01-05 |        426363 | kraken        |
|      14 |    843 | /nics/hdf5/1.6.10/cnl2.2_gnu4.4.1_par/hdf5-1.6.10/src/H5detect | faheymr  | 2010-01-05 |        426365 | kraken        |
|      15 |    843 | /nics/hdf5/1.6.10/cnl2.2_pgi9.0.1_par/H5detect                 | faheymr  | 2010-01-05 |        426368 | kraken        |
+---------+--------+----------------------------------------------------------------+----------+------------+---------------+---------------+
```

Figure 7: ALTD jobs table example.

## 5. Bibliography

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., et al. (1999). LAPACK Users' Guide (Third ed.). *Society for Industrial and Applied Mathematics.*

*Aprun* . (n.d.). From http://docs.cray.com/cgi-bin/craydoc.cgi?mode=Show;q=;f=man/alpsm/10/cat1/aprun.1.html

Balay, S., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., et al. (2008). *PETSc Users Manual.* Argonne National Laboratory.

Cray. (n.d.). *Using Cray Performance Analysis Tools*. From http://docs.cray.com/books/S-2376-41/

Gropp, W., Lusk, E., & Skjellum, A. (1999). Using MPI, 2nd Edition, portable Parallel Programming with the Message Passing Interface. *MIT Press In Scientific And Engineering Computation Series, Cambridge* , 395.

J. Borrill, J. C. (2005). Integrated Performance Monitoring of a Cosmology Application on Leading HEC Platforms . *International Conference on Parallel Processing: ICPP.*

John, L. (1999). *Linkers and Loaders.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Levine, J. (1999). *Linkers and Loaders.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Malony, S. S. (2006). The TAU Parallel Performance System. *International Journal of High Performance Computing Applications, SAGE Publications* , 20(2):287-331.

Mohr, B. (1999). TOPAS - Automatic Performance Statistics Collection on the CRAY T3E. *T3E, Proceedings of the 5th European SGI/Cray MPP Workshop.*

Papi. (n.d.). From http://icl.cs.utk.edu/papi/.

Rew, R. K. (1990). NetCDF: An Interface for Scientific Data Access. *IEEE Computer Graphics and Applications* , Vol. 10, No. 4, pp. 76-82.

Solchenbach, F. J. (1996). VAMPIR: Visualization and Analysis of MPI Resources.

Staples, G. (2006). TORQUE resource manager. *SC 06 PRoceedings of he 2006 ACM/IEEE conference* (p. 8). Tampa, Florida: ACM.

Tam, A. (2001). *Enabling Process Accounting on Linux HOWTO.*

Wheeler. (n.d.). From http://www.dwheeler.com/program-library/Program-Library-HOWTO.pdf

# 6. Appendix A : Overview on Libraries and Linkers Background

## a. Types of library

Program libraries can be divided into three types: static libraries, shared libraries, and dynamically loaded libraries (Wheeler).

- **Static libraries** are simply a collection of ordinary object files and they are installed into a program executable before the program can be run. Conventionally, static libraries name have the extension with the ``.a'' suffix.

- **Shared libraries** are loaded at program start-up and shared between programs. When the shared libraries are linked, the function symbols from the shared libraries are not copied in the executable and the size of the executable is considerably smaller than when using static libraries. The shared libraries name have the prefix ``lib'', the name of the library, and their extension are ``.so'' which can be followed by a period and a version number.

- **Dynamically libraries** (DL) can be loaded and used at any time while a program is running. DL are not really a different kind of library; instead, the difference is in how they are used by programmers. It involves a dynamically linked library loading and unloading at run-time on request. Such a request may be made implicitly at compile-time or explicitly at run-time. Implicit requests are made at compile-time when a linker adds library references that include file paths or simply file names. Explicit requests are made when applications make direct calls to an operating system's API at runtime such as the use of use “dlopen”, “dlclose” and “dlsym”.

Generally, in the programming literature dynamic and shared libraries are confused and the difference is not clearly stated. The .so files are even sometimes called as dynamically linked shared object libraries. There is only one form of this library however it can be used in two ways:

a. Dynamically linked at run time but statically aware. The libraries must be available during compile/link phase. The shared objects are not included into the executable component but are tied to the execution.
b. Dynamically loaded/unloaded and linked during execution using the dynamic linking loader system functions.

In this paper, we refer the first way as a shared library and the second way as dynamically library.

### b. Overview on linking

Usually, the operation of the linker is invisible to the programmer since it is automatically a part of the compilation process. The compiler compiles each source file to assembler and then object code, and links the object code together, including any needed routines from libraries (application or system library). Generally, the last step in compiling a program and just before the creation of the executable is to run the GNU linker(ld). It combines a number of objects and archive files, relocates their data and resolves symbol references. For more details, about the linker, we refer to (John, 1999).
At the linkage step, it is easier to track static and shared libraries than the dynamic one, since for DL, the library can be loaded during the runtime. Indeed, the linker needs to know which static and shared libraries are used by adding the -l or -L options with the library name in the linkage. Moreover, for the shared library and dynamic libraries, the dynamic linker/loader( ld.so) is called to load the libraries needed by a program, prepares the program to run, and then executes it. For a successful execution, some environment variables are needed such as LD_LIBRARY_PATH and LD_PRELOAD. The environment variable LD_LIBRARY_PATH is a colon-separated set of directories where libraries should be searched for first, before the standard set of directories while environment variable LD_PRELOAD lists shared libraries with functions that override the standard set.

Dynamically linked libraries used to introduce overhead during the execution of the program, however architectures such Cray XT systems recently developed new features of the Cray Linux Environment with scalable dynamic libraries which makes the performance impact negligible with dynamic libraries.

## 7. Appendix B: Alternatives

To the best of our knowledge, no tool has been developed for the explicit goal and objectives presented above. 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, ALTD is not intended to track all the function calls but rather the library 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 ALTD 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).

Is the following useful at all to explain how shared/dynamic abilities can be used to track stuff?

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".

The dynamic linker can be influenced into modifying it's behavior during either the programs execution or the programs linking. Examples of this can be seen with the ld (unix) manual page [1]. A typical modification of this behavior is the use of the **LD_PRELOAD** and **LD_LIBRARY_PATH** environment variables. The previously mentioned variables adjust the executables linking process by searching for shared libraries at alternative locations and by forcefully loading and linking libraries that would otherwise not be loaded and linked.