Franklin Job Completion Analysis

Hwa-Chun Wendy Lin, Yun (Helen) He, Woo-Sun Yang
National Energy Research Scientific Computing Center (NERSC)

ABSTRACT: The NERSC Cray XT4 machine Franklin has been in production for 3000+ users since October 2007, where about 1800 jobs run each day. There has been an on-going effort to better understand how well these jobs run, whether failed jobs are due to application errors or system issues, and to further reduce system related job failures. In this paper, we talk about the progress we made in tracking job completion status, in identifying job failure root cause, and in expediting resolution of job failures, such as hung jobs, that are caused by system issues. In addition, we present some Cray software design enhancements we requested to help us track application progress and identify errors.

KEYWORDS: XT4, *aprun*, *aprundat*, *aprunrpt*, apsched, NHC, syslog, SMW logs, SEC, CMS, Torque, job failure analysis

1. Introduction

The NERSC Cray XT4 machine Franklin has been in production for more than 3000 users since October 2007, almost a year after the XT4 was released in November 2006. In the early days of Franklin, we experienced a good share of system wide outages (SWOs). The good news is that after the 2.1 upgrade in December 2008 [J. Craw et al 2009, Y. He 2009]and a large scale I/O upgrade in March 2009, Franklin has become very stable, with system wide disruptions being mostly link inactives, some of which were attributed to environmental issues during the introduction of new systems.

SWOs are obviously disruptive to running jobs. It's easy to identify jobs that were lost to SWOs, but challenging to explain why other jobs failed. From the time a user reports a job failure until the problem is identified, there are usually several iterations of acquiring information from the user, checking system logs for hints, combining problem reports to find commonality, and searching the bug database for existing, similar bugs. This process involves users, consultants, system administrators, Cray support personnel, both on-site and remote, and sometimes developers.

Early on, gathering problem descriptions was tedious and challenging. There was not much users could tell us except their jobs got terminated—typically, there were no error messages shown on the output. Initially, we had to copy various system logs, including syslog and console logs, to hosts where consultants could access them. Consultants thus took on the responsibility of providing initial problem descriptions. They noticed that one of the frequently seen messages was out of memory (OOM), and asked that this type of message be written to user output. Over time, more and more error messages were written to user output files, and timestamps and identifiers were also added to error messages. The user output now is a useful information source in understanding how jobs run.

Understanding and reporting to DOE how Franklin jobs completed is a task that did not appear to be practical initially. In 2008, N. Cardo at NERSC started a systematic study of job completions, by enhancing the batch epilogue to search for known error message patterns in user output and to report *aprun* exit codes from the process accounting logs. The study was presented at CUG 2008 [N. Cardo 2008] In this paper, we describe a job completion reporting system that is similar, but with refined pattern searching and exit code processing. In addition, it makes use of system logs to further disambiguate job exit status.

In April 2009, NERSC formed a project team to study Franklin job completions. There are two goals for the team. One is to come up with a reporting system that has definite answers to whether a job ran successfully or failed, and if it failed, why. It may be a rather ambitious goal, as for some jobs, we may not be able to provide such answers without diligent studies. The second goal is to identify system issues that cause user jobs to fail, with the primary focus on hung jobs, and to work with Cray to fix them.

The team has compiled a list of possible system issues that cause user jobs to fail. (More in Section 4). In summary, user jobs can fail due to hardware failures in SeaStar links (an SWO), MOM (the batch job execution host) nodes, Lustre service nodes, including Data Virtualization Service (DVS), or compute nodes. User jobs can also fail due to system or system environment problems, such as out of memory (OOM), over use of /tmp (which is effectively memory), kernel bugs, portals bugs, over subscription of batch spool filesystem, or user authentication difficulty. In addition, user jobs can just hang, i.e. they don't appear making progress executing code, for some mysterious reasons that we are trying to identify.

In this paper, we first discuss the reporting system, then discuss in depth the two dominant system issues, including hung jobs.

2. Understanding Job Completion Status

At NERSC, we manage batch jobs using the combination of Torque and Moab. When a batch job exits, the Torque server generates an E (Exit) record in its accounting log. On the record, there is "Exit_status". The field name suggests it's the exit status of the job, when it is really the return code of the last command the batch script ran. Technically, it's the exit code of the script interpreter, and most interpreters exit with the status of the last command they ran.

XT batch jobs mostly launch applications to compute nodes via the *aprun* command. To understand how XT jobs complete is much more than looking at the batch job exit code—a successful simple *ls* command after an *aprun* could mask off the fact the application failed. We need to look in other places to find out how the applications actually completed. The sources for additional information we've identified are user output, *aprun* exit codes, and system logs.

2.1. System/software error messages on user output

We mentioned in the Introduction that with more and more system and application execution error messages written to it, the user standard error file (stderr) has become a good source to find out what happened to applications. These messages are typically intermixed with the user's own output lines at the beginning and in the end. How do we identify them? Short of applying data mining (log analysis) principles, the best we can do is build a known patterns list, and check each output line against this list for a match.

It's quite a challenge to come up with such a list. We first tried to locate published message catalogues and software error tags (prefixes). Unfortunately, we found only a couple, one for Cray Compilation Environment (CCE) runtime errors, the other PathScale runtime errors. The bulk of the patterns came from consultants' visual inspection of user output files, either when they received problem reports or when they ran out of more interesting things to do. We also defined "catch-all" patterns to match "suspicious" messages. We studied these messages for a while before deciding whether to ignore the messages or add new patterns to the list. This is an ongoing process, and the list grows over time. The catch-all patterns approach works because Cray has started to tag error messages. The most rewarding catch-all patterns so far are:

```
aprun: Apid
[NID <nid>]
```

There are often multiple patterns describing the same type of errors, such as out of memory (OOM). We make up labels (or categories) for error groups, not individual patterns. See Appendix A for the current list of known patterns and their labels. Here is a summary of the labels:

Compiler runtime	CCERUNTIME, PATHRUNTIME, SHAREDLIB
Message passing	MPIENV, MPIABORT, MPIFATAL
I/O	MPIIO, PGFIO, FILEIO
Jobs	JOBWALLTIME, JOBCOPY
Applications	APRESOURCE, APEXIT, APWRAP, NIDTERM, APNOENT, APEXEC, APDVS, APCONNECT
Signals	SIGTERM, SIGSEGV, SIGOTHERS
Portals	PTLSYS, PTLUSER
Miscellaneous	OOM, DISKQUOTA, IDENTRM, NOBARRIER, NODEFAIL, PERMISSION

One of the problems with using error messages is that they are not always available. Some users redirect their stderr files to private locations. We did a quick study and found about 15 percent of *aprun* stderr files were redirected. Moreover, the output for interactive jobs goes to the terminal, and there is no stderr file per se. We had opened a design bug with Cray for providing a way for *aprun* to save the content of stderr. But we stopped pushing the issue once we realized the volume of possible messages for large node count jobs and the possibility of scrambling the ordering of messages for *aprun* commands that merge stderr in with standard output (stdout).

Typically, we find multiple known patterns for a failed job. How do we decide which one to use to categorize the job? There is a hierarchy to the patterns in terms of the real cause for the fault. For instance, the "initiated application termination" message (label NIDTERM) is usually accompanying PGFIO or MPIFATAL, among others. As a result, this label is low on the totem pole. The current hierarchy was established through studying several days' worth of collected outputs. We believe we've come up with a reasonable hierarchy, but it is still subjective, based on knowledge and experience. We started an effort to quantify the process, but don't have anything conclusive yet.

On the other hand, even if we are able to derive a perfect hierarchy, there is still a problem. Patterns are grouped in such a way that the ones in the same group, such as APEXEC and MPIIO, don't show up for the same *aprun*. However, a lot of jobs launch *aprun* multiple times. One *aprun* may complete successfully, one terminate with APEXEC, and yet another with MPIIO. Which category should this job be assigned to? One possibility is that we get down to the application level and report success/failure ratio in terms of nodehours, i.e. how many nodehours are consumed by applications that ran successfully or that failed.

2.2. Application exit codes

The running of XT applications is controlled by the *aprun* command. One of this paper's authors, H. Lin, developed a tool called *aprundat* to collect jobs' *aprun* information as logged in the ALPS apsched log. This tool was described in a paper presented at CUG 2009 [H. Lin 2009]. She further expanded the data collection to include command line and exit code. Fetching the command line from syslog for an *aprun* is straightforward—all it takes is to find the syslog *aprun* record with the same *aprun* id (*apid*). On the other hand, it's more involved to find its exit code. After locating the *aprun* process id (*pid*) as stored in syslog, one has to look up in the process accounting log of the job's MOM.

The exit code of an *aprun* reflects that of the application, most of the time. As documented in the *aprun* man page:

If all application processes exit normally, aprun exits with zero. If there is an internal aprun error or a fatal message is received from ALPS on a compute node, aprun exits with 1. Otherwise, the aprun exit code is 128 plus the termination signal number of an application process that was abnormally terminated, or the aprun exit code is the exit code of an application process that exited abnormally.

Aprun's overriding of application exit codes is a serious problem when application exit codes are not available anywhere else. We opened a design bug with Cray asking that *aprun* syslogs its internal error and avoid exit code 1, as 1 is a popular exit code. The solution Cray has agreed to includes adding up to four application exit codes and/or signals to the *aprun* exit record (*apsys*), as well as logging *aprun* internal errors to syslog. Until this solution is implemented, we will still use *aprun* exit codes in the job categorization process, fully aware of its implications.

2.3. System logs

In addition to knowing whether jobs ran successfully or failed, we also want to know, for failed jobs, whether the failures were results of system problems. The primary use of system logs is to differentiate system caused errors from user caused errors. For example, a job that got time limited (label JOBWALLTIME) could be a result of SWO, when effectively the job was still accumulating wall clock time. We've observed, on Franklin, when the batch system restarted after an unfortunate event such as SWO, all previously active jobs were first requeued, then got purged when MOMs came up. Thus we created a label called JOBREQUEUE for such jobs to document jobs lost to SWO. Without it, most of these jobs would fall into the JOBWALLTIME category.

At the present time, the only information we harvest from the Torque server logs is job requeues. Another piece of information that may be useful is job deletion records, for they tell us whether user or root deleted these jobs. In the MOM log, we plan to look at copy failure related messages to decide whether the failure was due to some filesystem issue, which caused difficulty in creating spool file, or just a case of specified, nonexistent path. We also plan to search for "no such user" messages, as an indication of LDAP look-up failures.

As mentioned in the Introduction, jobs die when their MOM nodes crash. We don't really know how to identify these jobs yet, but we plan to find an answer for it.

Among all system logs, the most interesting ones are those available on the System Management Workstation (SMW): console log, netwatch log, and consumer log. However, those logs contain numerous types of messages in different formats about nodes, Lustre, HSN, portals, etc., it's challenging and time-consuming to analyze and interpret them.

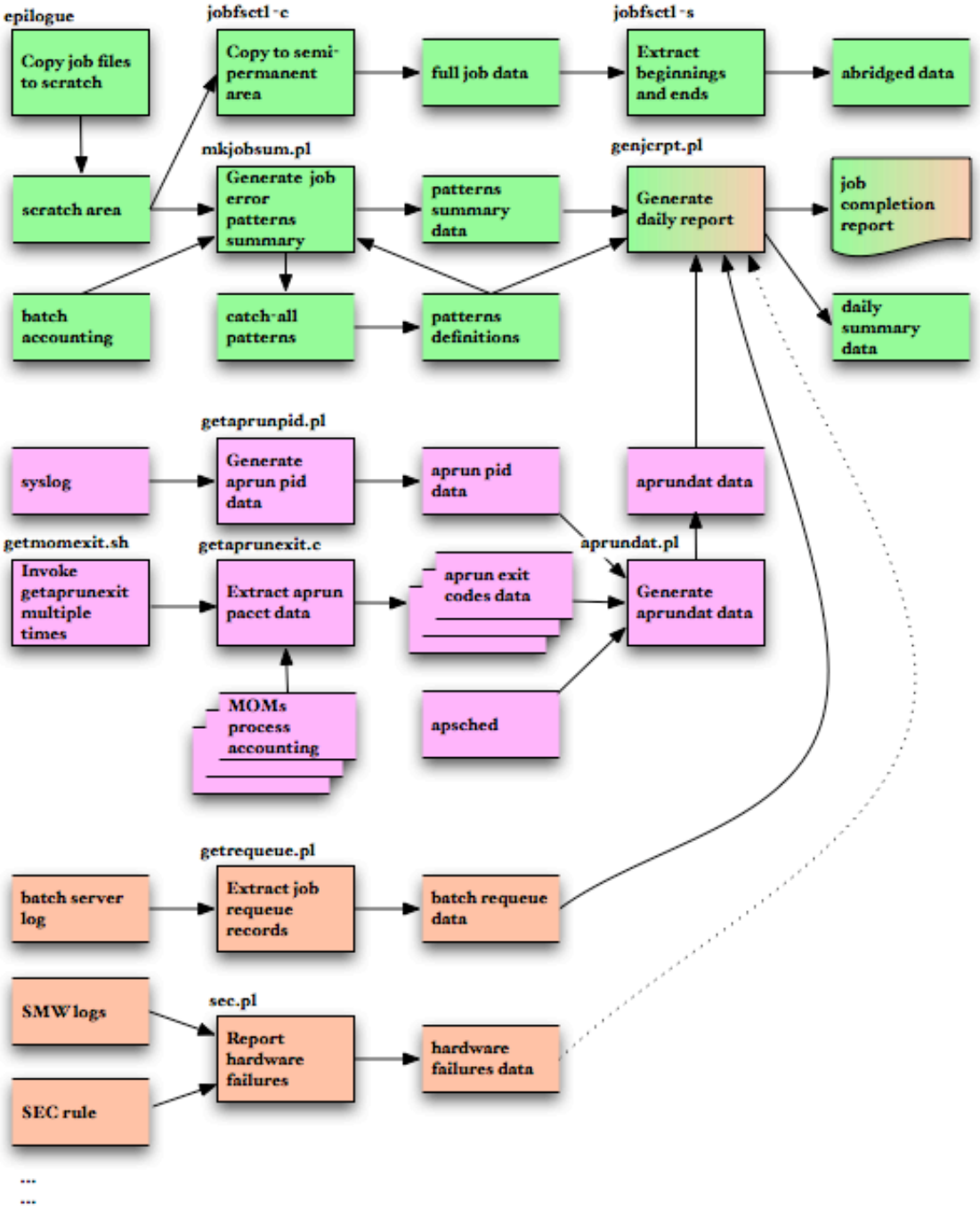
J. Becklehimer and C. Willis of Cray and Oak Ridge National Laboratory staff developed a set of Simple Event Correlator (SEC) rules to track primarily hardware failures by correlating and interpreting these messages. This work was presented at CUG 2007 [J. Becklehimer 2007], while Jim Brown's online documentation is helpful in understand and working with SEC [J. Brown 2003].

Cray recognized the usefulness of SEC, and released a field notice (FN #5495) recommending it. NERSC Cray support personnel put together SEC rules to monitor additional failures including Bugs (Cray bugs) and LBUGs (Lustre bugs) on external Lustre servers.

At the moment, we receive e-mail notifications about various events detected by SEC and other Cray monitoring scripts. The challenge is how to incorporate the information into the job completion report framework.

3. Implementation Phases

Due to the complexity of this report generation system, the implementation was done in phases. Below is a flow chart that shows how all the pieces fit together, and which phase each piece belongs to. The green boxes (the top three rows) describe the first phase implementation; pink (the middle three), second; and orange (the bottom part), third. The daily report generation keeps evolving and becomes more complicated as we progress in phases, with additional types of data to process.



3.1. Phase I implementation

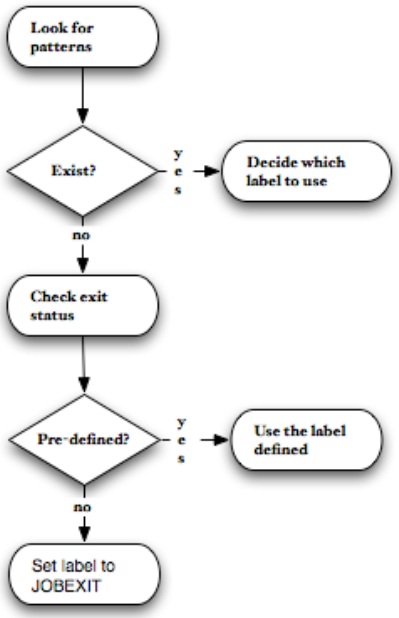
The epilogue processing is done after a job script has completed and before the job leaves the batch system. What goes in the epilogue is very site specific. At NERSC, after writing job statistics to stdout, we copy job files: the script file, stdout, and stderr, to a scratch area. Job files accumulate in this scratch area during the day. Early the following day, the *jobfsctl* Perl script copies all job files to a semi-permanent place, then sometime later, the script is invoked again, but with a different option, to create abridged files. Small files are copied verbatim, but large files are elided in the middle.

What's the purpose of creating abridged data? We run a script to check job output file size periodically and delete jobs with huge output (> 100 MB). But out of control applications still manage to generate GB files. Output files are useful for both systems personnel and consultants to debug job issues, so we try to save them for as long as possible. Unfortunately, huge files eat up limited storage

quickly. We theorize that important error messages typically show up at the beginning and in the end on batch output, so we don't really need to save files in their entireties. We are currently studying this hypothesis

The *mkjobsum* Perl script post processes the output files to find all known patterns for jobs run on the reporting day. The script gets the job list from the batch accounting log, where it also extracts exit statuses found in jobs' E (exit) records. For each job, the summary line has job sequence number, followed by exit status, followed by all known patterns found. For each unmatched line, *genjcrpt* runs it through predefined catch-all patterns, if a match to one of the catch-all patterns is found, the script notifies project team members, who then decide whether to add new patterns. All patterns are defined in a Perl "include" file. The patterns list is constantly changing, but the *genjcrpt* script stays static.

There were no other information sources available during phase I, so the report generation script *genjcrpt* relied solely on the job summary information provided by *mkjobsum* to assess job completion statuses. Because of this, we didn't categorize jobs that we couldn't find fault for as success, we used NOKNOWNERR instead. The decision making process for this phase is fairly simple-minded, and shown below.



The pre-defined labels are NOKNOWNERR for job exit status 0, JOBSTART (MOM could not start the job.) for "-2", JOBPROLOG (The prologue script returned error.) for "-1", SIGTERM for 143 and 271, SIGSEGV for 137 and 265.

There are three parts in the daily report. The first part is a summary of job completion statuses. It shows all nonzero categories, and for each category, how many jobs and the calculated percentage. The "Cause" column reflects how we view the cause: user ("U"), system ("S"), or either ("US"). We realize sometimes some temporary system situation could cause applications to fail. The cause assignment doesn't reflect this indeterminism.

The following is a sample output. This report is selected to show a variety of errors, it doesn't describe a typical day. We don't normally have APDVS, or STALENFS, or so many APNOENT failures.

Exit Status	Count	Percent	Cause
APDVS	1	0.0	S
APEXEC	2	0.1	U
APNOENT	36	1.7	U
APRESOURCE	12	0.6	U
CCERUNTIME	1	0.0	U
JOBEXIT	122	5.9	U
JOBPROLOG	4	0.2	US
JOBSTART	3	0.1	S
JOBWALLTIME	240	11.5	US
MPIABORT	3	0.1	U
MPIENV	4	0.2	U
MPIFATAL	7	0.3	U
NIDTERM	128	6.1	U
NOCMD	20	1.0	U
NODEFAIL	1	0.0	S
NOENT	48	2.3	U
NOKNOWNERR	1287	61.8	N/A
OOM	11	0.5	U
PATHRUNTIME	1	0.0	U
PERMISSION	1	0.0	U
PGFIO	41	2.0	U
SHAREDLIB	1	0.0	U
SIGSEGV	25	1.2	U
SIGTERM	57	2.7	U
STALENFS	27	1.3	S
XBIGOUT	1	0.0	U
Total	2084		

The second part shows statistics for causes.

Job Failure Statistics

Type	Count	Percent
No known err	1287	61.8
System	32	1.5
User/system	244	11.7
User	521	25.0

The third part provides a list of users who appear to be having problems running jobs. This was added when we noticed an extremely high count (> 250) of PGFIO jobs, and got alarmed. It turned out to be a user issue.

```
-------------
High Counts for Category+User
-------------
Category      User         Count
--------      ----         -----
APNOENT       userabc         10
APRESOURCE    userb            9
JOBEXIT       usercd          55
JOBWALLTIME   userdef         31
JOBWALLTIME   useref          19
NIDTERM       userfg          18
NIDTERM       userg           61
NOCMD         userhi           9
NOCMD         userjkl          8
NOENT         userklm         11
OOM           usermno          6
PGFIO         usernop         14
SIGSEGV       usero            8
SIGTERM       userp            5
…
```

The *genjcrpt* script also saves daily summary information to a file to facilitate job completion trend analysis.

3.2 Phase II implementation

The real work for phase II was to enhance the *aprundat* script to include application command line and *aprun* exit code. As mentioned in section 2.2, the *aprun* exit code for an application is in its execution host's process accounting log, which means we have to find the *aprun* execution host and pid first. Process ids get recycled rapidly, each pid is only unique for a limited time period, so we need to also match up on the process end time.

The execution host and pid can be found in the *aprun* record in the syslog, while its end time is in the corresponding *apsys* record. Pairing *aprun* and *apsys* is nontrivial, because an *aprun* could span multiple days. Further, we discovered matching up the two end times, one in syslog, the other in process accounting, is also challenging. We had anticipated slight end time difference between the two, and put in a fudge factor to accommodate it, but were surprised to see that the difference could be several minutes. We took notes about this peculiarity, but do not have an explanation yet.

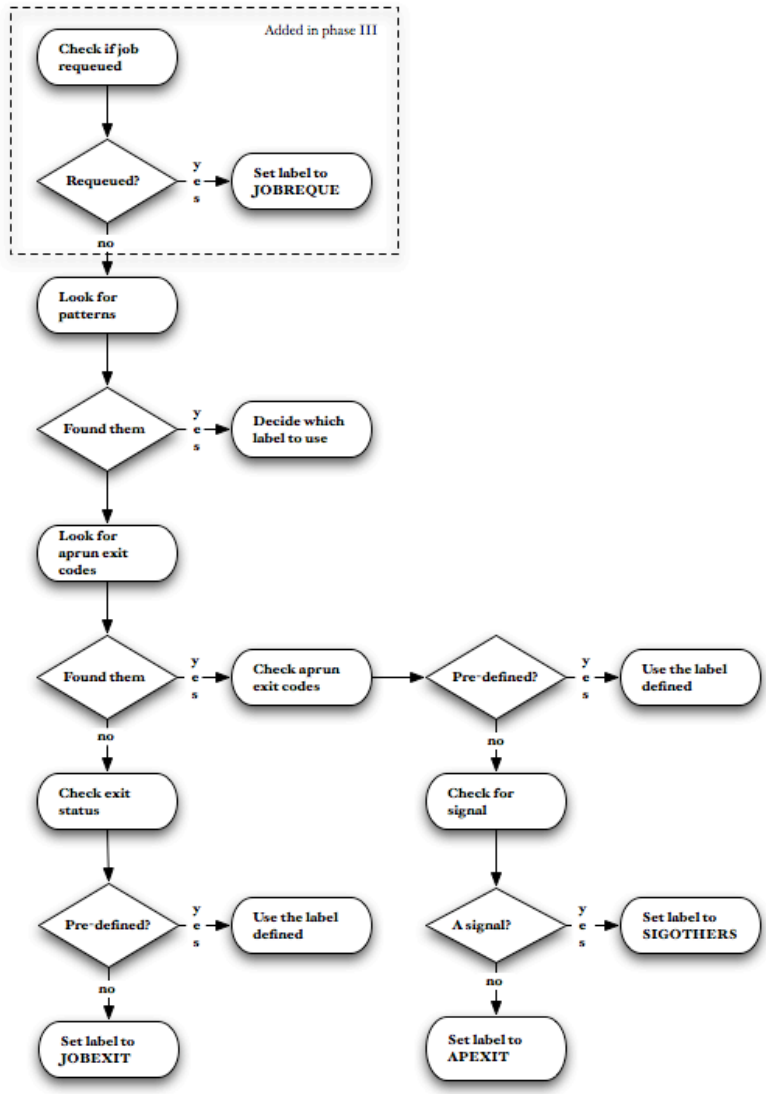
The original *aprundat* as reported by its report generation counterpart *aprunrpt* looked like this, with irrelevant fields skipped:

```
Job ID    …   User        Command/node list
6467851   …   abcdef      hostname/9031-9046
```

while the enhanced *aprundat* looks like this:

```
Job ID    …   User     exitcode   Command/node list
6467851   …   abcdef   0x0000     "aprun -n 64 hostname "/9031-9046
```

The decision making process in *genjcrpt* for category assignment, with additional *aprun* exit code information, is shown next.



The pre-defined labels for special exit codes are SUCCESS for 0, APRUNERR for exit code 1, SIGSEGV for 0x8b00, SIGTERM for 0x8f00.

The phase II daily report is very similar to that of phase I, but with more derived categories. The most important difference is that we retired the label NOKNOWNERR, and replaced it with SUCCESS.

3.3 Phase III implementation

The third phase implementation is still in progress. The only additional system log in use is the Torque server log, where we look for job requeue entries. The workflow is very similar to that of phase II, with an additional level shown in the box framed by dashed lines.

4. System Issues

As we mentioned in the Introduction, one of the job completion team's goal is to identify system issues that caused user jobs to fail, and to work with Cray to fix them. In general, failed jobs waste system resources. For system induced job failures, we also have to refund users. Even with the refund, jobs resubmitted go to the bottom of the queue, thus impact user productivity.

System related job failures we have observed usually fall into the following categories (no specific order of occurrences):

- System wide outages. The causes of such incidents could be Lustre nodes crashes, link failures, High Speed Network (HSN) congestion, and power issues, etc. Almost all jobs during a SWO would fail. Some jobs may survive during link inactives.
- Batch execution nodes (MOM nodes) crash. A MOM node could die due to hardware issues or when running out-of-memory. For example, some users running memory intensive applications (such as IDL, htar), combined with other's incorrect usage of running program executables without *aprun*, could cause a MOM node to run out of memory. All batch jobs executed from a failed MOM node would fail. In the early days, a MOM node fail would also cause a SWO due to the need to reboot the system. Being able to warm boot service nodes helped tremendously in reducing the number of failed jobs. NERSC also worked on the separation of login nodes from MOM nodes, so that out-of-memory login nodes would not affect running batch jobs.
- LDAP related user authentication failures. See more details in Section 4.1.
- One or more nodes used by an application have hardware failures.
- "Sick" nodes left by previous jobs. The subsequent jobs would fail if some of the nodes were in "unhealthy" state. Some jobs would fail due to out-of-memory errors, and some jobs would hang.
- Hang jobs/applications. See more details in Section 4.2.
- Aprun awaiting barriers. Many of these errors are caused by user issues, such as serial jobs running out of walltime, or when concurrent apruns running in the background without a "wait" at the end. But there may also be system related issues we don't understand yet.
- /tmp filled. /tmp used by a previous application is not cleared, causing the subsequent applications after that to fail due to out-of-memory. NERSC has set the maximum /tmp size to 512 MB, and are looking into options for cleaning /tmp before a new application launches.
- /var filled. Applications fail when writing stderr/stdout into spooled directories. We run a periodic check on user output files, and terminate jobs if they've produced 100 MB of data.
- Program environment related issues. One example was loading xtpe-quadcore explicitly caused subsequent "module list" to fail. This problem has been fixed.
- Portals bugs related issues. One of the examples in this category was jobs getting "MPICh PtlEQPoll event->ni_fail_type error (OTHER EQ handle): This is likely due to an unresponsive node on the system" error. Jobs normally exited after 5 min. Initially identifying these nodes via console log "beer" messages of "cpu xxx has been unresponsive for 240 seconds" and setting them to "admindown" helps to not schedule these nodes to future jobs. This problem has been fixed with a portals patch.
- Portals related system issues. Jobs get "PTL_NAL_FAILED" or "PTL_PT_NO_ENTRY" error messages.
- Lustre IO related problems. Accessing an existing file/directory sometimes generates "input/ouput error".
- DVS server failures. This affects jobs running from file systems that are projected to the compute nodes via DVS.

The two dominant system issues we have studied in depth are LDAP lookup failures, and hung applications. The good news is the magnitude of these problems has dramatically decreased.

4.1 LDAP lookup failures

LDAP stands for Lightweight Directory Access Protocol. NERSC has a center-wide LDAP server where Franklin gets user and group information. The first lookup problem we noticed was that the local Name Service Cache Daemon (nscd), an LDAP client, died of SIGABRT regularly. As a result, users complained that they could not log in. We opened a Bug for the problem, but in the meantime, we put in a periodic check and restart nscd when necessary.

The second encounter we had was when users started making inquiries about "Identifier removed" error messages. The error was seen when users were trying to access files, issuing *cd*/*ls*/*cp*, also when applications were trying to open files. This message is the *perror()* text for errno 43 (EIDRM), which Lustre can return for file meta data operations when the secondary group upcall done by *l_getgroups* fails. The error can happen at any stage in running jobs, as well as in interactive sessions.

One instance of these errors was traced to timeouts contacting the site LDAP server (Bug 742050). A more common situation was when *l_getgroups* failed to look up a valid UID, before doing a search for secondary group

membership. Bug 751388 was submitted on the failure to look up valid UIDs, which was associated with "l_getgroups: no such user xxx" messages in syslog.

In the process of debugging the EIDRM problem, it was realized that *l_getgroups* bypasses *nscd* in looking up multiple group membership, which can cause unanticipated overhead at a site with thousands of groups. Bug 751387 was submitted requesting that *l_getgroups* be modified to make use of *nscd* group caching.

Local research on the UID lookup failure bug scenario was conducted to determine if the failures were on the XT/*nscd* side or on the LDAP server side. The NERSC LDAP support personnel helped determine that the errors were not occurring on the LDAP server. Rather, it appeared that *nscd* threw away information received from LDAP after a cache timeout.

The *nscd* daemon and associated libraries were upgraded to newer versions (FN #5615) to attempt to address the issue, but that did not appear to make difference in terms of EIDRM errors.

Eventually Cray changed the *nscd* configuration to set the shared attribute for user and group lookups, which allows *nscd* clients to search the *nscd* database directly without having to build a socket connection each time. This appears to have substantially reduced the number of EIDRM errors, although some "no such user" errors are still recorded in syslog.

Further updates to *nss_ldap* and *nscd* have been provided by Cray and will be tested to see if they improve the user/group lookup situation.

The label we assigned to describe the "Identifier removed" error message is IDENTRM. The JOBCOPY ("Unable to copy") error can happen for the same reason, if MOM can not create a spool file, even before starting the job script. This is different from the copy error that happens at the job end, when MOM has trouble copying a spool file back to user directory. In the latter case, spool files are available for retrieval at a later time.

LDAP lookup failures can also happen at batch job starting time, because MOM needs to authenticate users before running the batch script. The batch Exit_status for such jobs is –2, and we put them in the JOBSTART category. At NERSC, failures are seen occasionally in the prologue, where the script checks to make sure users still have allocations left. The account information is fetched from the NERSC accounting system via LDAP. An error exit can be a result of a failed LDAP lookup, or of an account's running out of allocation. Jobs failed due to a prologue issue have the Exit_status of -1, and we label them JOBPROLOG.

4.2 Hung jobs/applications

We use the term "hung jobs," as well as "hung applications," to describe jobs that are not making visible progress on output. A job can hang because an application it runs is hung in the middle of execution, or it can hang from the beginning. A hung job is terminated by the batch system when it exceeds time limit, or by the user when they sense something is wrong

User code can cause a job to hang because of a deadlock in MPI communication, an infinite loop, or a perpetual wait for job input. This type of job typically has generated some amount of data in output, and rerunning them with a higher debug *aprun* option "-D" usually helps detect the coding problem.

On the other hand, job hangs caused by system issues are not reproducible, and we usually receive user reports about "previously running applications now hung". This type of hung job normally has nothing on output at all, and the "-D" also doesn't generate more information. The analysis of hung jobs appeared to sometimes implicate hardware, but we believed the primary causes were software, due to portals issues, Lustre issues, or dirty nodes left by previous applications.

The hung jobs situation is much improved since the 2.1 upgrade, when Cray introduced the node health checker (NHC) utility. [CrayDocs S-0014-22 & S-2425-22] The node cleanup utility is launched to the compute nodes, if an application exited "unorderly", i.e. *aprun* did not receive exit information from all the compute nodes assigned to the application. When the utility finds a "sick" node, it sets it to "admindown" so that the node will not be used by subsequent jobs. The definition of "sick" evolves as more and more checks are added.

In addition to providing a node health checking utility, Cray has also responded to our portals/Lustre bug reports and supplied fixes for them. With the upgrade of 2.2, fixes for important bugs are in. Even though we don't receive too many user complaints about hung jobs now, we still see a fair number of jobs that terminated for walltime limit (JOBWALLTIME). Until we know more, we cannot be certain the JOBWALLTIME is now entirely a user issue.

Several bursts of user application hangs were reported last year. Error messages associated with one such burst looked like: "aprun: Caught signal terminated awaiting barrier, sending to apid xxxx" (label NOBARRIER) followed by walltime exceeded. Lustre error messages found in console logs were something like this:

"[c5-4c1s0n2]Lustre Error 31373:0:
mdc_locks.c:586:mdc_enqueue())ldlm_cli_enqueue: -4".

Setting affected nodes admindown helped alleviate the hung job situation. Further, Cray was able to trace this problem to a portals issue related to "transmit credit accounting." A patch was installed on Franklin last September, and the official fix went in with the CLE 2.2 UP02 upgrade.

However, after the patch went in, we still saw the NOBARRIER error, even for serial programs, and were puzzled. This barrier is a startup synchronization barrier

that *aprun* expects to receive. Because only applications that make use of the Process Management Interface (PMI), such as MPI, shmen, UPC, or CAF, can supply such information, all non-PMI applications, when they are terminated due to the wall time limit, will always get this message. Cray proposed to drop the "awaiting barrier" from the NOBARRIER message, to eliminate the confusion. But we figured that if *aprun* could set an alarm to time out the barrier wait for only PMI applications, we could actually detect hung applications this way. The problem is now how to differentiate PMI versus non-PMI, which is even harder now with the shared libraries support. Cray offers to implement the alarm, but an application has to pass in the desired alarm time via an environment variable. A site can develop a wrapper to set it on behalf of applications, or users can set it. (Bug 755008)

Another burst of application hangs appeared to be associated with the console message "The mds_connect operation failed with -16" (bug 756028). Compute nodes with this error lost connection to the Lustre file system (the access to /scratch hangs) and their apinit process was hung. Cray investigation indicated the attempt to connect to MDS failed due to EBUSY, when the MDS node hit an LBUG. Rebooting these nodes recovered access to Lustre. A change to the Lustre "group_acquire_expire" setting for MDS from 15 to 60, then to 240 seconds seemed to be able to avoid this problem, which was related to intermittent failures by the MDS node when talking to LDAP.

The introduction of the node health checker is a big step toward enhancing the XT productivity by increasing the rate of successful jobs. However, more node abnormality checks are needed. For example, one of the contributing factors to sick nodes is memory related. The /tmp on the compute nodes is a tmpfs filesystem, i.e. it's effectively memory. If it doesn't get cleaned up after an application, it can cause the next memory bound application to fail for insufficient memory. Even worse, it can trigger the Linux out of memory (OOM) killer to start killing processes. Important daemons, such as apinit, unless they are protected, can get killed and cause communication breakdown and result in a hang situation. Other unwelcome memory consumers include runaway processes, and unreleased slab memory. In general, the node health checker should admindown a node that doesn't have documented available memory.

Detecting and handling hung applications is very high on our priority list. Besides the barrier re-implementation and additional checks in the NHC, we also explore the possibility of having MPT initialization routines print a completion message, and of running a short system health check program before starting applications. These are longer term tasks that require more thorough analysis and planning. We hope the new barrier implementation and expansion of NHC functionality will be sufficient.

5. Beyond Reporting

The primary motivation for providing an elaborate job completion analysis tool is to fulfill one of our contractual requirements to DOE. Not too surprisingly, though, the job information, either raw or summarized, turned out to be useful other ways.

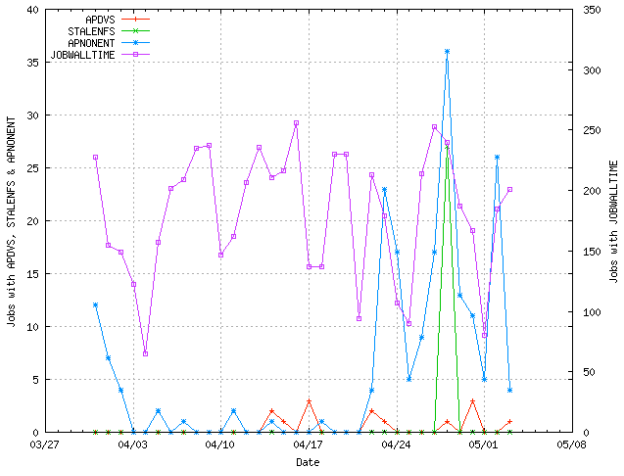
The raw data, i.e. job files, are being used to spot system wide issues. For example, one user reported these error messages after a Torque upgrade:

```
Cannot connect to default server host '<host>' - check
pbs_server daemon.
qsub: cannot connect to server <host> (errno=111)
Connection refused
```

Grepping saved output uncovered more of the same, except there were various `host` names, none of which was the "real" PBS host. We found an issue with the new Torque!

The third table in the report labeled "High Counts for Category+User" is being used to promote proactive user services. As a rather drastic case in point, a consultant noticed that one user registered over 2000 failures in the JOBEXIT category and investigated. The problem turned out to be a mistake in removing a directory without specifying the "-r" option. We were able to contact the user in a timely basis.

The daily summary data are being used to analyze the error trend. Below is a plot for all the data we have, with four categories we are interested in: JOBWALLTIME, APDVS, APNOENT, and STALENFS.



Notice that the JOBWALLTIME is plotted at a different scale—its numbers are much larger, relative to the others. Since we are only curious about the trend, it doesn't really matter. The interesting part shown in this graph is that STALENFS appears to contribute to the high APNOENT, which makes sense.

We expect more use of the job information when we start generating more summary data, about job size and compute resource use, etc.

6. Conclusion

The Cray systems hardware and software have improved a great deal over the three years we have had Franklin. Not only is Franklin more stable, but now have error messages available to the user to summarize how jobs ran. This information is a great source for debugging problematic jobs, either by users or consultants. In addition, we are able to leverage the information, and use it as a building block, to further understand and analyze Franklin jobs as a whole.

Cardo's work in job completion analysis as presented at CUG 2008 [N. Cardo 2008] paved the way for the work described in this paper. One important difference between the two is that the analysis and category assignment is no longer performed in the epilogue script, at the end of each job run. The epilogue now only saves job files to a pre-defined location, where the files accumulate during the day and are post-processed later. Because the processing is done in bulk once a day, it can be as elaborate as needed, and it can be repeated when new patterns are identified and added in.

We've already built a good collection of known patterns which cover a wide range of errors, including compiler runtime, file I/O, portals, etc., but we still manage to find new ones. Thanks to Cray for tagging their messages, which permit us to implement the idea of catch-all patterns. Hopefully, Cray will eventually tag all the messages they have control over, to maximize the likelihood of unseen messages being caught. In fact, it'd be perfect if Cray could make available messages catalogues for Cray specific system components, such as ALPS and portals. Furthermore, we'd also like to see similar messages be consolidated, to keep the patterns table from growing out of hand.

7. Future Work

We are currently in the early stage of the third report generation implementation phase. We plan to look in various system logs for information to use to disambiguate the "US" (caused by either system or user) errors. The largest US category has always been wall time exceeded (JOBWALLTIME). We know some users checkpoint their applications or launch an application multiple times, and their jobs are designed to run into wall time limit. But there are others that hit time limit unexpectedly, due to a bug in the code, or an issue with some aspects of the system: nodes, HSN, Lustre, etc. How do we tell them apart? We suspect system logs can shed some light.

In addition to finding ways to distinguish the problem source for hung applications, we continue the effort of eliminating hang sources. On the system side, we've been talking with Cray about incorporating some sort of node health check in *aprun* before starting applications, as well as installing an alarm when setting up the initial barrier in *aprun* for MPT applications. We've also been looking into helping users debug hung applications. We heard about Abnormal Termination Processing (ATP) and friends a few months ago, and are looking forward to the Cray Debugging Support Tools (DST) tutorial at CUG 2010. [B. Moench 2010]

We expect to make changes to some existing data collection methods. The report generation framework is well structured, built on independent data collection modules. The framework was designed to accommodate additional data collection modules easily.

But we realize that a couple near future Cray system changes are likely to affect how we collect data. One change is the availability of the Cray Management System (CMS). CMS, a successor to Mazama, according to J. Schildt in his CUG 2009 paper, is "a framework that integrates hardware state information and software environment to provide monitoring and administration functionality for Cray XT system." [J. Schildt 2009] Progress has been made and Schildt will chair a BOF session on CMS HOWTO at CUG 2010. We may want to take advantage of the CMS, but don't yet know the magnitude of changes as a result.

The other change to an existing job completion data collection module is to happen when *apsys* starts to syslog up to four application exit codes and signals. We should then be able to bypass process accounting logs and get application exit codes directly from *apsys* records in syslog.

We mentioned that we are studying to see whether abridged data are as good as the originals, hoping that we don't lose error messages by saving only the beginning and ending lines. We have come to appreciate the ample information found in user output and would like to keep it for as long as we have room. The smaller the files are, the more they can be kept. If the result of the study doesn't support our hypothesis, we may decide to manually trim these huge files.

As for the report itself, we plan to also report compute resource usage by node-hours, and statistics based on job size. The additional information will be added to the daily summary file to allow more in-depth job data analysis.

Acknowledgments

This work was supported by the Director, Office of Science, Division of Mathematical, Information, and

Computational Sciences of the U.S. Department of Energy under contract number DE-AC02-05CH11231.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

The authors would like to thank two Cray employees: Steve Luzmoor and Rita Wu. They responded quickly to our queries and helped us gain better understanding of issues. Luzmoor further assisted with problem analysis, and also authored and tracked many Cray Bugs.

The authors would also like to thank Tina Declerck of NERSC for her work in hung job investigations.

References

Jeff Becklehimer, Cathy Willis, Don Maxwell, Josh Lothian, David Vasil, "Real Time Health Monitoring of the Cray XT3/XT4 Series Using the Simpe Event Correlator (SEC)," Cray User Group Conference Proceedings, 2007.

Jim Brown, "Working with SEC—the Simple Event Correlator," http://sixshooter.v6.thrupoint.net/SEC-examples/article.html, 2003.

Nicholas P. Cardo, "Detecting System Problems with Application Exit Codes," Cray User Group Conference Proceedings, 2008.

Jim Craw, Nicholas Cardo, Yun (Helen) He, Janet Lebens, "Post Mortem of the NERSC Franklin XT4 Upgrade to CLE 2.1," Cray User Group Conference Proceedings, 2009.

Yun (Helen) He, "User and Performance Impacts by Franklin Upgrades," Cray User Group Conference Proceedings, 2009.

Hwa-Chun Wendy Lin, "Understand Aprun Use Patterns," Cray User Group Conference Proceedings, 2009.

Jason Schildt, "System Administration Data under CLE 2.2 and SMW 4.0," Cray User Group Conference Proceedings, 2009.

CrayDoc S-0014-22, "Application Cleanup by ALPS and Node Health Monitoring," <docs.cray.com> 2009

CrayDoc S-2425-22, "Cray XT System Software 2.2 Release Overview," Section 2.8 <docs.cray.com> 2009

FN #5495 - XT System monitoring utility

FN #5615 – SLES10 glibc and nscd update 2009

About the Authors

Wendy Lin is a Systems Analyst at NERSC, and the system lead for the Franklin job completion project. E-mail: hclin@lbl.gov.

Helen He is a High Performance Computing Consultant at NERSC, and the User Services Group point of contact for Franklin. Email: yhe@lbl.gov.

Woo-Sun Yang is a High Performance Computing Consultant at NERSC. E-mail: wyang@lbl.gov.

Appendix A: Patterns and Labels

Pattern	Label
[NID <nid>].*Connection timed out	APCONNECT
Apid <apid>: DVS server failure detected: killing process	APDVS
Apid <apid>: cannot execute: exit(<exit code>)	APEXEC
Application <apid> exit codes:	APEXIT
aprun: file.*not found	APNOENT
Claim exceeds reservation's	APRESOURCE
exceeds confirmed width	APRESOURCE
*aprun*wrapper:	APWRAP
lib-####	CCERUNTIME
Disk quota exceeded	DISKQUOTA
Cannot send after transport endpoint shutdown	FILEIO
Input/output error	FILEIO
Identifier removed	IDENTRM
Unable to copy file	JOBCOPY
PBS: job killed: walltime.*exceeded limit	JOBWALLTIME
application called MPI_Abort	MPIABORT
MPICH.*out of unexpected buffer space	MPIENV
MPI_MSGS_PER_PROC is not sufficient	MPIENV
PTL_EQ_DROPPED	MPIENV
Fatal error in MPI	MPIFATAL
ROMIO-IO level error:	MPIIO
MPI-IO level error:	MPIIO
initiated application termination	NIDTERM
aprun:.*awaiting barrier	NOBARRIER
Received node failed or halted event	NODEFAIL
Machine Check error	NODEFAIL
ALLOCATE.*not enough memory	OOM
OOM killer terminated this process	OOM
Process ran out of memory	OOM
unable to acquire enough huge memory	OOM
Fortran runtime error:	PATHRUNTIME
PGFIO-F	PGFIO
PTL_NAL_FAILED	PTLSYS
PTL_PT_NO_ENTRY	PTLSYS
PTL_NO_SPACE	PTLUSER
PTL_.*VAL_FAILED	PTLUSER
PTL_.*SEGV	PTLUSER
error while loading shared libraries	SHAREDLIB
LIBSMA ERROR:.*PtlGetAddRegion	SHMEMATOMIC
exit signal.*Segmentation fault	SIGSEGV
aprun:.*Caught signal Terminated	SIGTERM
Stale NFS file handle	STALENFS
fixout: Extra huge output trimmed	XBIGOUT
No such file or directory	NOENT
Permission denied	PERMISSION
Command not found	NOCMD