Reliable Computation Using Unreliable Components

Joel O. Stevenson, Robert A. Ballance, J. P. Noe, Suzanne M. Kelly, Jon R. Stearley
Sandia National Laboratories*
Mike Davis, Cray Inc.

SAND-2013-3309C
Unclassified, Unlimited Release

April 23, 2013

Abstract

Based on our experiences over the last year running 32K–64K core simulations on Cielo, a Cray XE6, we present strategies that we are using to enable large, long-running simulations to make predictable progress despite platform component failures. From an application perspective, complex systems like Cielo have multiple sources of failures or interrupts that combine to make the system appear unstable to users (file system issues, node dropouts, transient network errors, system time, *etc.*).

As the system experts continue to drive toward the resolution of the root causes of component failures, the application developers and users of the system can mitigate some of the issues by employing scripting mechanisms that trap and identify failures and then recover where possible. We will discuss the component failures that we have experienced and identify those where recovery has been possible. We will also discuss the scripting mechanisms for error trapping and identification, recovery, and re-launch of applications.

1 Perspective

Researchers, engineers, and system administrators are working actively to ensure successive generations of large-scale HPC machines are increasingly resilient to failures. Adopting the terminology from Snir *et al* [17], we define resilience as *"The collection of techniques for keeping applications running to a correct solution in a timely and efficient manner despite underlying system faults."*

This paper focuses on user-level resilience techniques that build atop system and application level resilience mechanisms. In our experience, most users need to submit, run, and successfully complete $M \geq 1$ jobs in order to gather the data needed for their study. Within this paper, we treat a *job* as the unit of batch queue submission. Each job may include multiple individual runs (*e.g.* 'apruns', in the case of CLE). Let us adopt the term *study* for the set of jobs that require completion. Each study consists of multiple jobs, and each job consists of at least one run, as depicted in Figure 1. (We will define τ, δ, and R in Section 5). Some users manage a long-duration study, managed as a *train* of submissions, having many checkpoints and many restarts. Others need to complete a large collection of jobs (or trains of jobs) running independently. Within a single job, there will be one or more runs.

Large systems are engineered artifacts, subject to engineering failures both internal and external. Henry Petroski [16] argues that the engineering that goes into a Cray (or other HPC) can usefully be treated as a "hypothesis" to be tested in the real world. In that real world, events that kill a job can arise from many sources: hardware, software, system reboots,

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

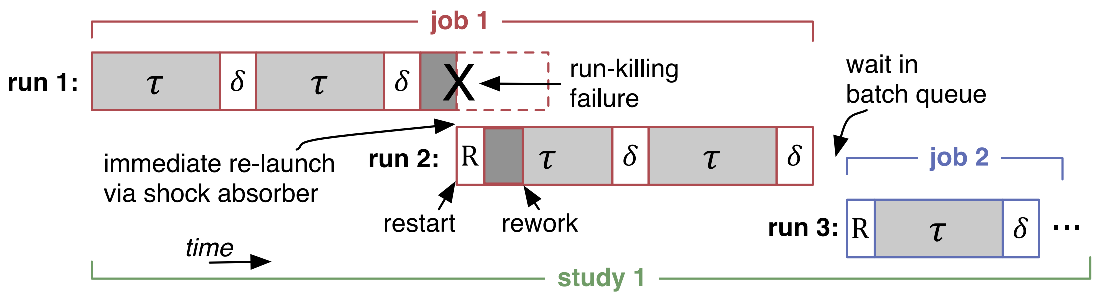

Figure 1: Jobs, runs, and studies

file system glitches, and even user errors. Section 2 outlines a taxonomy of these sources.

Since runs have a propensity to fail, Section 3 introduces the "shock absorber" pattern of job submission scripts. This useful design pattern [11] shields a single job execution from perturbing events. The purpose of the shock absorber script is to maximize the productivity within each job, by handling run terminations properly. A shock absorber can be crafted to handle failures arising both from the run (e.g. *aprun*) and from the environment (e.g. *moab*). Sandia has been successfully using this pattern since the early days of Red Storm, and aspects have recently been incorporated into the "resilience" feature of *aprun*. Section 4 discusses one user's (Joel's) experience in using the shock absorber pattern to run jobs on Cielo, a 1.33 PF Cray XE6 located at Los Alamos National Laboratory (LANL) and operated jointly by LANL and Sandia National Laboratories under the Alliance for Computing at Extreme Scale (ACES) partnership on behalf of the DOE NNSA/ASC program.

What happens when the shock absorber doesn't work? Section 5 presents some work done by ACES and at Sandia to understand rework after a job is killed, and to minimize the work lost. Section 6 provides an introduction to using "grace" signals that warn applications of impending terminations. Section 7 briefly discusses the tools at your disposal to try to figure out what went wrong. This is an important step, since it may lead to your making the "shock absorber" smarter and more resilient. Finally Section 8 provides some high-level recommendations for your consideration.

2 Resilience and Error

The demands of large scale, and exascale, computing are leading to renewed interest in resilience techniques. The frequency of workshops[8; 10; 17] attests to the interest in making HPC systems and applications more resilient to failures. John Daly and his collaborators[7; 13] have investigated application resilience, largely from the perspective of checkpointing behavior and rework.

However, as systems grow in size, the applications grow in size and time as well. Barney Maccabe[15] of Oak Ridge National Laboratory once characterized these as "constant-time applications:" make a machine 16 times faster, and the user will double the resolution in 3 dimensions, double the run time, and the same problem will take just as long to complete as before. This is why the notion of a *study* is key: it is the entire study that has to be efficiently completed, and no matter how resilient the system, the appetite for cycles will likely outstrip the resilience mechanism.

There are four different sources of job execution-killing errors: two in the engineering domain, and two in the human domain.

In the engineering domain, runs (and jobs) may succumb due to (i) a localized problem (such as a sync-flood CPU error) or (ii) due to a system-wide event. In either case, there are several contributing factors.

First, individual runs may fail due to a localized fault, such as a hardware, software, or application issue. If p_j is the probability of an isolated run failure, we know that p_j is a function of three factors:

$p_{j,hw}$	the probability of a hardware component fault given the job size
$p_{j,ssw}$	the probability of node-level system software fault given the job size
$p_{j,app}$	the probability of application fault given the job size and input parameters

Second, there's always the possibility that as your job is running, the entire system fails. Let p_r be the probability of system reboot when your job is running. Then p_r is a function of four components:

$p_{r,hw}$	the probability of a system-level hardware fault
$p_{r,ssw}$	the probability of a system-level software fault
$p_{r,ext}$	the probability of an external fault (e.g file system, power) that kills the system, and
$p_{r,sched}$	the probability that your job is killed by scheduled system time[1].

In the human domain, runs (and jobs) may succumb due to (i) user error, or (ii) operator error. Let p_{user} denote the probability of a user error (including errors in complicated inputs such as mesh and materials specifications) and let p_{admin} denote the probability of system operator error.

Assuming that your study of M batch submissions is always running, then, the probability of failure-free workload during M job runs is given by

$$p_{success} = ((1 - p_j)(1 - p_r)(1 - p_{user})(1 - p_{admin}))^M$$

As M increases, this becomes a (vanishingly) small number.

The engineering numbers and the system administration numbers can be empirically estimated given the right data from the system. However, correlating job failures back to root (or near-root) causes continues to be difficult and time-consuming. Error messages can, and should, be designed with eventual consumption by automated processes in mind.

[1] Sites use various scheduling strategies to deal with system reservations. On Cielo, rather allowing nodes to go idle prior to a system time reservation, we prefer to let them run as long as possible. This means that not all jobs will receive their full increment of time. In addition, jobs may hang, or may be killed due to preemption.

This observation ties back to Petroski's argument — if any complex engineering artifact is actually a hypothesis about how to build a system, then the system designers need to include sufficient information about how the system is performing to assess whether or not the hypothesis holds! In our case, we know that CLE is wise enough to kill jobs when components fail, but the system analysts task of mapping from the component fault to the job actually killed is non-trivial. NERSC and Sandia have reported on tools to accomplish this task via scripting or Splunk[12; 18], but better reporting from CLE would make it easier to estimate p_j and its contributing factors. Section 5 presents an idea for measuring the total overhead of checkpoint/restart on a study, and some work done by ACES to minimize rework

3 The Shock-Absorber Pattern

Better designs, better software, better hardware — all will reduce the numerical value of these probabilities, but not to zero immediately. The shock absorber pattern evolved to add resilience at many levels (site, admin, and user) to whatever resilient job execution mechanism is present.

The basic idea in the shock absorber pattern is to wrap the execution launch command (e.g. *aprun*, *mpirun*, or *mpiexec*) inside a script that can both catch and handle system signals (if necessary) and also trap, assess, and handle execution errors. Figure 2 shows a conceptual view. Note that the pattern

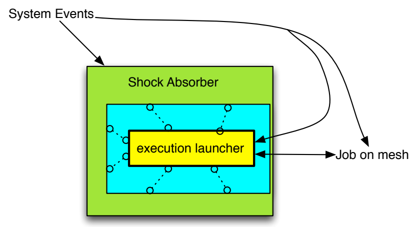

Figure 2: Shock Absorber

tries to manage events coming from both the system to the script, and events arising from the execution command.

```
set_signal_handlers();
outcome = UNKNOWN;
while (nodes_available(...) && time_left(...) &&  outcome == UNKNOWN) {
    try (run_job(...)) {
        catch (all_done()) { outcome = SUCCESS; break; }
        catch (fatal_error()) { outcome = FATAL; break; }
        catch (restartable_error()) {
            // set up for restart & continue loop
        }
}
switch (outcome) {
    SUCCESS: declare_victory(); break;
    FATAL: bad_things_happened(); break;
    GOT_SIGNAL:  signal_happened(); break;
    UNKNOWN:  ran_out_of_nodes(); break;
}
```

Figure 3: Pseudo-code for shock absorber

A more detailed, but still schematic, version of the code appears in Figure 3. Converting this code to a maintainable script can be messy, but it is straightforward. A more difficult problem is to define and encapsulate the recoverable errors in an extensible and sharable format, so that as new errors appear, they can be automatically propagated into the user's environment.

3.1 History of the pattern

Sandia has been successfully using this pattern since the early days of Red Storm (2005). The code eventually evolved into *ryod*.

ryod was designed as a wrapper around *yod* to address issues of resilience. With Release 1.2 of XT3 software, users experienced a high failure rate (10–20%) for *yod* launches. Reasons for failure varied, but the most commonly observed were:

- Load error: an individual node became unresponsive as *yod* tried to load an app onto it. In some cases, a subsequent *yod* on the same set of nodes, excepting the failed node, would succeed;
- Node failed event: an individual node failed, and an app was running on it at the time. In some cases, a subsequent launch of the app on the same set of nodes, excepting the failed node, would permit the app to progress further.

The duties of *ryod* were to:

- Detect the most common errors in loading, launching, and execution prevalent on the system;
- Execute recovery measures where possible, with the objective of achieving a successful load, launch, and execution;
- Log errors to a system-wide database, in a format for easily generating summary reports;
- Be flexible enough to accommodate tuning by the user and administrator in the face of changing system conditions.

The fourth requirement continues to be important moving forward; local interpretation and handling of errors needs to be extensible and customizable, on both a site and per-application, job, or study basis.

3.2 Job Submission with the Shock Absorber

Jobs submitted using the shock absorber typically reserve several extra nodes so that if a node or blade falls out, the shock absorber can relaunch a new run without going back through the scheduler queue. Sizing the number of extra nodes is heuristic: smaller jobs may need to lock down fewer nodes for a given runtime. However, if every job reserves extra nodes, the overhead can add up. In addition, the shock absorber itself needs to maintain state concerning the number of extra nodes reserved, and the number used, so that it can detect when it runs out of nodes.

A feature that would be highly useful in the batch management system would be a call, executed from a service node, to request that additional nodes be added to the current partition. Such a call would enable the shock absorber to request new nodes as needed, rather than reserving them ahead of time. One could even envision ALPS doing this for us; when a node is failed in a partition, a new one could be added!

The shock absorber also needs a notion of the time remaining in the batch job, since it might not make sense to attempt a restart when there is little time remaining. Lawrence Livermore National Laboratory supports a library[9] that enables any application to query its time limits.

Finally, we recommend configuring (in the scheduler) and handling (in the script or the application) a grace signal that is sent to the job in order to warn of impending wallclock limits or system shutdowns. The signal is can be used to gracefully terminate the application after writing a restart file. This technique minimizes rework when a job has to be restarted. Section 6 provides further details.

3.3 What gets handled?

On Cielo, the generic shock absorber (written as a *bash* script) handles three key cases: (i) `ec_node_failed` (ii) `ec_node_halted`, and (iii) `nem_gni_error_handler`. If not on a failed node, we manually tie up the node (to take it out of the partition). This error can be detected when the failed node has reported a Gemini error; by trial and error we have found that it is best to avoid relaunch on the node that reported the Gemini error.

3.4 Resilient Aprun

CLE 4.1 introduced "resilience features" to the *aprun* command[14]. The features allow *aprun*, when called with the `-R` option, to retry a run if the job receives either a node failed (`ec_node_failed`) or a node halted event (`ec_node_halted`). In either case, the user is able to specify the number of nodes that are consumed from the partition in each cycle.

The resilience aspect of *aprun* is an important step, but the initial deployment has two limitations that are problematic. First, the two errors to be handled seem to be built in. To handle local errors, like the Gemini errors on Cielo, one will still need to run the shock absorber. Second, the initial deployment of resilient *aprun* does not provide (internally) for any job cleanup following a failure. So if cleanup/restart actions are needed, it will be necessary either to perform that work from within the application based on the value of the environment variable `ALPS_APP_RELAUNCH`, or to resort to a shock-absorber. There are also potential file system and application relaunch interactions with the node health checker[14].

4 Experience with the Shock Absorber

From an application perspective, complex systems like Cielo provide multiple sources of slowdowns, interrupts, and failures that combine to make the system appear unpredictable and unstable to users. It is very difficult to get good application throughput on a large simulation that requires months on the mesh without an effective strategy to combat node dropouts, transient network events, *etc.* One needs a purposeful strategy to achieve reliable computation using unreliable components. It won't just happen on its own. When wait times in the scheduling queue can typically be several days or longer, fully utilizing your turn by maximizing application time on the mesh is very important. You will not make simulation progress if you go to the end of the queue line every time your large job is ejected from the mesh due to a failure or interrupt.

The shock absorber is a strategy that maximizes application availability and efficiency by trapping and identifying system failures and then recovering where possible (i.e. re-launch the run within your job). The shock absorber scripting mechanism provides a finer level of job control that improves app availability and work throughput and moves the simulation forward despite external interrupts. We will discuss our experience using the shock absorber to harness a 64K core long-duration (970 hours) single study, managed as a train of job submissions, having many checkpoints and many restarts.

As shown in Figure 4 , it required 70 jobs to achieve 970 hours wall-time for our 64K core job. In a perfect world it would take 40 jobs (24 hours/run). In fact, 40% of the jobs completed without error in 24 hours; 43% of the jobs experienced a preventable or recoverable error; and 17% of the jobs experienced an unrecoverable error. It is notable that almost half (43%) of the jobs experienced a preventable or recoverable error. If you do not employ a shock absorber to re-launch the application within your allocation, you will give up a tremendous amount of "mesh time", significantly slowing the progress of your job.

The notable metric in Figure 4 is the average run time before and after deploying the shock absorber. The first 48 jobs were performed without the shock absorber; the remaining 22 jobs were performed using the shock absorber. The average run time before deploying the shock absorber was 11 hours, 42 minutes; the average run time increases by almost $2x$ to 21 hours, 41 minutes after deploying the shock absorber.

Alternatively, consider the average daily runtime accumulated by the top 3 users on Cielo during a recent Capability Campaign as shown in Figure 5. Can you tell which user was using a shock absorber?

5 Rework

Periodic checkpoint/restart is the most widely used failure mitigation technique. Given a series of runs in a study using checkpoint/restart (via shock absorber or chained job submissions), some amount of rework is performed each time a job starts from checkpoint, corresponding to computations which were not saved due to an intervening failure. Although there is much in the literature on how to compute the checkpoint interval that minimizes rework (and time to solution) [3; 6], there is very little in the literature about what checkpoint intervals are are used in practice, and the corresponding efficiency of the technique[7]. Note again Figure 1 which illustrates the relationship between runs, runs, and studies. Within a given run, the amount of time between checkpoints is τ, the time to write a checkpoint is δ, and the time to restart from a checkpoint is denoted by R[6].

Although users know their run-specific checkpoint intervals, the information is not collected across users; keeping such a list up to date would be burdensome and provide limited value. It is possible that checkpoints are being written too frequently, resulting in longer time to solutions due first to non-optimal intervals, but secondly due to filesystem contention leading to system outages. In the worst case, users respond by writing checkpoints more frequently, leading to a vicious cycle with those responsible for filesystems holding the bag.

The fact that many jobs write checkpoints at regular intervals provides an opportunity. Now suppose that applications emitted a `syslog()` message at the start and end of writing checkpoints. A simple way to expose this information would be for rank 0 of synchronous checkpoint applications to `syslog()` a message indicating when they begin writing a checkpoint, and again when they stop writing the checkpoint.

Administrators could easily identify which runs were checkpointing at any time (which is often crucial information during filesystem problems!), and actual checkpointing practice could be studied including total I/O time, estimated rework time, and checkpoint-interval efficiency on both a per-run and system-aggregate basis.

Communicating via `syslog()` may not be the optimal mechanism, but they are simple, available today, and can be exposed to system-wide log-analysis tools. The resulting syslog load should be small (if it is not, that itself will indicate a problem that should be addressed), and it should provide a signal that

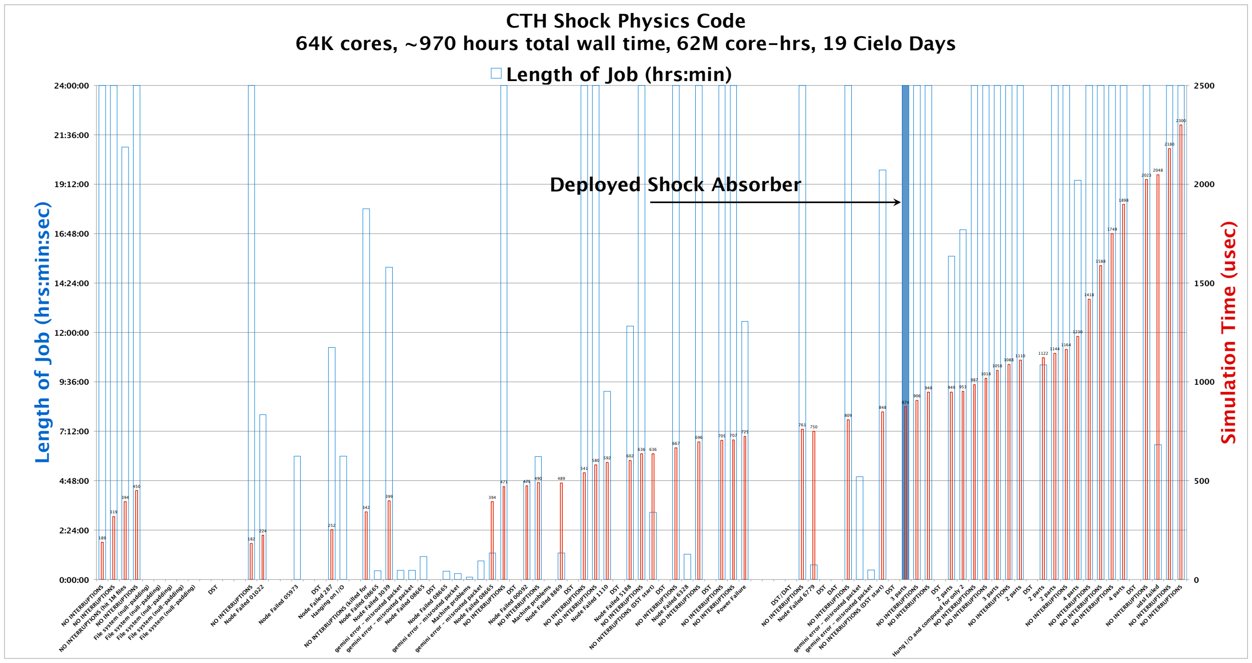

Figure 4: Progress of 64K processing element job

system administrators can use to more proactively and confidently identify runs that may be hung. For example, a run that writes hourly checkpoints for 15 hours and then writes nothing for 5 hours is suspect.

Regardless of the mechanism used, a log of checkpoint start and stops for all runs on the system would provide significant operational and research value.

6 Signals and Grace

One goal of high-performance computing is to maximize simulation throughput. One aspect of this is to increase the efficiency of the simulation job workflow. Analysts typically are constrained by a maximum wall clock time policy associated with their HPC machines. Allocated time slots are typically 24 to 96 hours in length and a full study, which can be many hundreds of hours, is therefore comprised of many aggregate jobs and runs. Each job writes checkpoint/restart sets to the file system during the simulation and dependent jobs read the checkpoint/restart sets to resume calculations. In many cases, the amount of wall clock time needed for a job is dependent upon its characteristics and is not easily quantifiable, e.g., if the user(s) require adaptive mesh refinement or for each iteration to have a specific residual, then the simulation time can fluctuate greatly. If the simulation environment contains the ability to process grace signals, which are used at Sandia and Los Alamos National Laboratory (LANL) to communicate that the wall clock limit will soon be reached, then the analyst can maximize simulation progress by saving the state of the simulation just prior to wall clock time and utilize all available compute cycles. Another benefit with signal handling is that it provides the HPC machine administrators a mechanism to gracefully shut down the machine while minimizing collateral damage on running simulations.

During the 3rd quarter of 2012, Sandia standardized on sending a signal to running simulations with the purpose of notifying them that their available run time will end shortly. This follows the LANL standard for signaling the end of application time. The POSIX

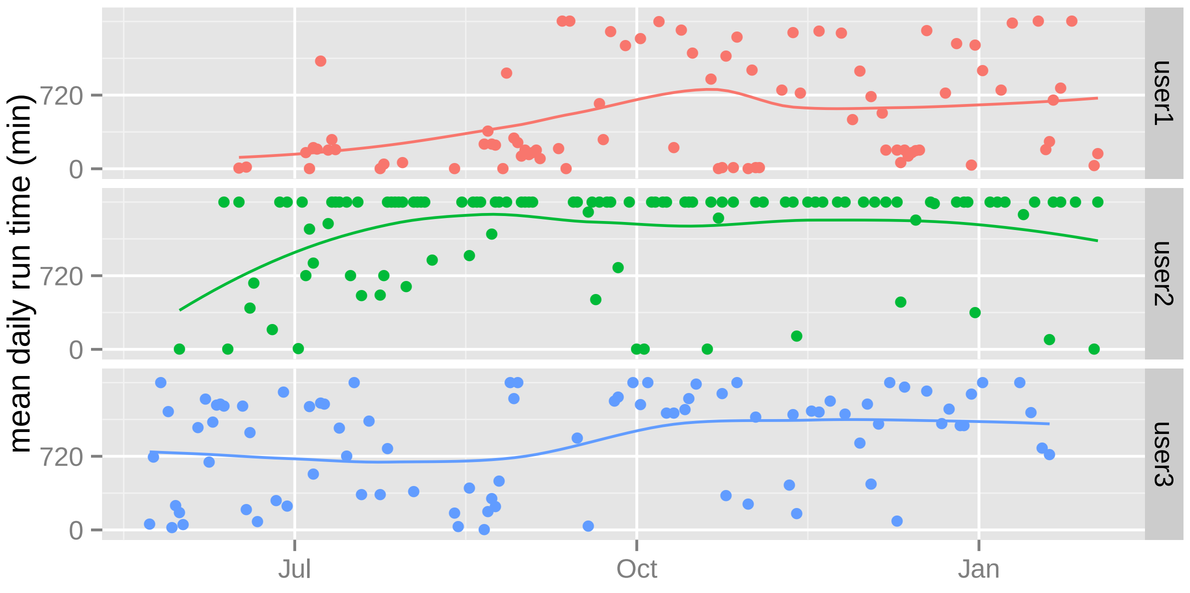

Figure 5: Top 3 users, average daily run time. User2 was using a shock absorber

Signal 23, also known as SIGURG, is the default signal to sent for this notification[1].

For the majority of cases, this notification will be used to warn the simulation that its wall clock limit is near. For some cases, however, this signal can be used by the administrators to nicely stop running simulations in the event of a system emergency. This notification will be automatically sent 10 minutes prior to the wall clock limit being reached. All of the Sandia-supported queue managers allow overriding this default setting within the job submission script or on the command line for cases where 10 minutes is not sufficient warning. It is also possible to specify an alternate signal.

Lawrence Livermore National Laboratory (LLNL) provides a related, but alternative mechanism by way of their "get remaining time" facility. This library call allows the application or shell programmer to query the amount of time remaining in the application's time box, and to take defensive actions such as checkpointing as appropriate[9].

In a rich environment, both methods can coexist, and each has specific use-cases distinct from the other. LLNL is successfully using both on its clusters.

7 When All Else Fails

Not all failures have an obvious signature. Job hangs may be difficult to distinguish from long computational times. Application-provided periodic status messages throughout a computational cycle can provide assurance that forward progress is being made. But the question then arises as to what to do next. Should the job be restarted? Was it perhaps a hung I/O operation or transient network congestion? Or was it an application hang that must be corrected manually before the job can continue?

Identifying hung runs can be challenging and various strategies have been proposed to ease the task[4]. In practice, users have a good sense of which files should be written when — whether they be application logs or checkpoints. The regularity of file operations provides a hint by which to identify hung runs. Shock absorbers can be programmed to look for these periodic update messages. For example, the job submitter can identify the name of a log file to monitor, along with an expected update interval. If the "dead man" timer expires, the application can be terminated.

Once the potential hang is detected, various tools can be used to capture the job state before a signal is sent to kill it. On Red Storm, the tool was called *fast_where*. It was a simple shell script that could gather the results of traditional debuggers' "where" command and coalesce them quickly (i.e. fast). The results were text-based which theoretically allowed scripts to look for outlier/rogue MPI ranks in different libraries (e.g. MPI or I/O) than the remaining ranks. In practice, human eyes were necessary to review the output to determine whether the job needed manual intervention or could be restarted. In CLE systems, the comparable tool is *STAT* (Stack Trace Analysis Tool)[2]. *STAT* is very similar to *fast_where* in that it coalesces stack traces from all processes in a running job and places them in a file for later viewing with *STATView*. Once collected, the job can be resumed. If it was a transient error, the job will make progress and the *STAT* output can be ignored. If a hang has occured, *STAT* can be used a second time which might help determine if the same signature is found. At this point the job should not be resumed. The submitter can review the *STAT* graphical output and decide upon the next course of action.

CLE systems also provide the Abnormal Termination Processing tool (ATP)[5]. Rather than setting a dead man timer, an environment variable can be set to trigger a stack trace, a la *STAT*, from all MPI ranks when the job terminates abnormally. When the job's time allocation is about to expire, perhaps with a hang in progress, the workload manager will send a signal. This signal is considered an abnormal termination and the stack traces will be collected. There is a risk that the application will trap the signal and the stack traces will not provide the true location of the hang. Of course, ATP can always be enabled to capture traces of any failure. Due to some compatibility concerns, Cray does not enable ATP by default.

8 Conclusions

Human anticipation and imagination will always outrun the capabilities of our HPC systems. No matter how robust and resilient, errors will happen, and design constraints will be exceeded. The wise user, then, will adapt and employ techniques like the ones described here to their own application, system, and circumstance. However, we'll close by summing up our recommendations for user, for system administrators, and for the system developers.

8.1 Recommendations for Users

- Plan for failures right from the start. Use shock absorbers! They work.
- Keep meticulous records. Knowing the errors that affect *your* application, their timing and their frequency, will help you to adjust your shock absorber.
- Insist that your application team provide you with good diagnostics. Without clear messages, including well-documented exit codes, you can't improve your response.
- Enjoy life when the system handles the failures for you. As resilience techniques improve, your "real" job should get easier.

8.2 Recommendations for System Admins

- Implement grace signaling. It takes extra effort to send a grace signal to each job, especially when the grace period can be job-specific. However, this is a matter of scripting.
- Find the root cause for all errors, and maintain a library of recoverable errors with their app-level signature.
- Be wary of system changes that affect job submission scripts. Users employ complicated scripts, so changes to the user environment may have deep ramifications.

8.3 Recommendations for System Builders

- Ensure that improvements like `aprun -R` are extensible with local knowledge.Adding local knowledge is the essential growth path for making smarter systems.
- Work to reduce component and system-level outages
- Assess logging data and make sure that job-level failures (and their cause) can be easily identified and tracked. For example, in this paper, we'd like to find all the job-killing events, and which job they killed. That data takes multiple joins over the data in the current deployment, but we're pretty sure that there's a place where system software knows both the error, and the job to be killed.
- Keep building bigger, smarter, and more robust systems!

9 Acknowledgements

The authors want to thank Anthony Agelastos (SNL) for his work supporting grace signals at SNL, the Cray/ACES team at LANL and Sandia, and many coworkers for their contributions to this work.

References

[1] Anthony M. Agelastos, Robert A. Ballance, and Mike E. Davis. Sandia signal handling primer. In preparation., May 2013.

[2] Dorion C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. Stack trace analysis for large scale applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Long Beach, CA, March 2007.

[3] Sarala Arunagiri, John T. Daly, Patricia J. Teller, Seetharami Seelam, Ron A. Oldfield, Maria Ruiz, and Varela Rolf Riesen. Opportunistic checkpoint intervals to improve system performance. Technical Report UTEP-CS-08-24, 2008.

[4] Robert A. Ballance and Nathan A. DeBardelen. Non-invasive job progress monitoring: The MoJo application monitoring tool suite. In *LCI Conference on High-Performance Clustered Computing*, Pittsburgh, PA, March 2010.

[5] Cray, Inc. Atp 1.6 man pages. `http://docs.cray.com/cgi-bin/craydoc.cgi?mode=View;id=sw_releases-0fxsp770-1351629007;idx=man_search;this_sort=title;q=;type=man;title=ATP%201.6%20Man%20Pages`, November 2012.

[6] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.

[7] J. T. Daly, L. A. Pritchett Sheets, and S. E. Michalak. Application MTTFEvs. platform MTBF: A fresh perspective on system reliability and application throughput for computations at scale. In *Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 795–800, 2008.

[8] John Daly, Bob Adolf, Shekhar Borkar, Hathan DeBardeleben, Mootaz Elnozahy, Mike Heroux, David Rogers, Vivek Sarkar, Martin Schulz, Marc Snir, and Paul Woodward. Inter-agency workshop on HPC resilience at extreme scale, 2012.

[9] Christopher J. Morrone Donald A. Lipari. Options for retrieving remaining time under moab. Technical Report UCRL-SM-228839, Lawrence Livermore National Laboratory, 2012. Updated, March 19,2012.

[10] E. N. M. Elnozahy, R. Bianchini, T. ElGhazawi, A.Fox, F. Godfrey, A.Hoisie, K. McKinley, R. Melhem, J. S. Plank, P. Ranganathan, and J. Simons. System resilience at extreme scale. Technical report, Technical report, Defense Advanced Research Project Agency (DARPA), 2008.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design.* 1995.

[12] Woo-Sun Yang Hwa-Chun Wendy Lin, Yun (Helen) He. Franklin job completion analysis. In *Proceedings of the Cray Users Group Meeting (CUG)*. National Energy Research Scientific Computing Center (NERSC), 2010.

[13] William M. Jones, John T. Daly, and Nathan A. DeBardeleben. Application resilience: Making progress in spite of failure. In *Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 789–794, 2008.

[14] Marlys Kohnke. Alps application relaunch. Revision 1.0 (modified 4/5/13), 12 2011.

[15] Arthur B. Maccabe. Personal communication.

[16] Henry Petroski. *To Engineer is Human: The Role of Failure in Successful Design.* Vintage, 1992.

[17] Marc Snir, Robert W. Wisniewski, Jacob A. Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Bill Carlson, Andrew A. Chien, Pedro Diniz, Christian Engelmann, Rinku Gupta, Fred Johnson, Jim Belak, Pradip Bose, Franck Cappello, Paul Coteus, Nathan A. Debardeleben, Mattan Erez, Saverio Fazzari, Al Geist Sriram, Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. In preparation.

[18] Jon Stearley, Robert Ballance, and Lara Bauman. A state-machine approach to disambiguating supercomputer event logs. In *Proceedings of the USENIX Workshop on Managing Systems Automatically and Dynamically (MAD)*, 2012.