

Application Resilience: Making Progress In Spite of Failure

William M. Jones
Electrical and Computer Engineering Department
United States Naval Academy[†]
Annapolis, MD, USA, 21402-5025
Email: wmjones@usna.edu

John T. Daly and Nathan A. DeBardeleben
High-Performance Systems Integration
Los Alamos National Laboratory[‡]
Los Alamos, MN, USA, 87545
Email: {jtd, ndebard}@lanl.gov

Abstract—While measures such as raw compute performance and system capacity continue to be important factors for evaluating cluster performance, such issues as system reliability and application resilience have become increasingly important as cluster sizes rapidly grow.

Although efforts to directly improve fault-tolerance are important, it is also essential to accept that application failures will inevitably occur and to ensure that progress is made despite these failures. Application monitoring frameworks are central to providing application resilience. As such, the central theme of this paper is to address the impact that application monitoring detection latency has on the overall system performance. We find that immediate fault detection is not necessary in order to obtain substantial improvement in performance. This conclusion is significant because it implies that less complex, highly portable, and predominately less expensive failure detection schemes would provide adequate application resilience.

I. INTRODUCTION

Cluster computing has been a viable supercomputing alternative for many different application domains for more than a decade. As the price-to-performance ratio of off-the-shelf computing and networking hardware has continued to decrease, making use of larger clusters to help solve computationally expensive problems has become extremely popular and rather common-place. The desire to improve model fidelity by running larger and more detailed scientific simulations is a constant in what has often become a race to build the world’s largest and most powerful cluster computers. While measures such as raw compute performance and system capacity are still at the forefront of our desired cluster characteristics, such issues as system reliability and application resilience have started to become increasingly important factors when evaluating overall cluster effectiveness.

As the size of a cluster increases, the system mean time between failure (SMTBF) tends to decrease inversely proportional to the number of components. Applications can experience an interruption in service due to such failures; however, *other events* can also cause an application to stop making progress. For example, a transient

error in the network or bug in system software could cause an application to hang while the associated cluster infrastructure remains entirely functional. This situation is often overlooked in favor of focusing on catastrophic node or network failures; however, novel system software stacks coupled with legacy parallel scientific applications deployed on modern cluster platforms push the envelope of reliability.

Much work has been done to improve system reliability, and therefore performance, by attempting to provide redundancy or other fault-tolerant technologies to keep the application from ever being interrupted. While this is a very interesting research area, it is not the focus of this paper. We accept that applications will inevitably be interrupted and are primarily interested in mitigating the overall impact on application and system performance in the face of these interruptions.

An important issue in addressing this impact is being able to determine whether an application is in fact making progress. Furthermore, we must analyze how quickly this determination needs to be made. In this paper, we present several possible techniques to monitor application progress and suggest several scenarios where each would be practical. Additionally, we provide evidence based on both analytical models as well as simulation that suggest that simple, less expensive application monitoring techniques can provide sufficient improvement in application throughput to obviate the need for more complex and intrusive methods. Furthermore, we demonstrate that application monitoring detection latencies need not be instantaneous; that detection times less than five to ten minutes provide only marginal benefits to both average job throughput as well as interrupted application execution time. Finally, we allude to scenarios where faster, more sensitive application monitoring would be required.

[†]This work was partially supported by the Naval Academy Research Council and ORN grant N0001407WRZOI02. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the Los Alamos National Laboratory, the US Naval Academy or the United States Navy.

[‡]This paper has been approved for public release under LA-UR-07-8150.

II. MONITORING APPLICATION PROGRESS

While there are obvious performance benefits to recognizing that an application has stopped making progress, it is not always clear precisely how to make this determination. The solutions can range from simple to complex and each has associated trade-offs. The methodology used will likely be tailored to the specific application and the system configuration.

Determining if an application is performing useful work can be very difficult. However, it may be possible to catch some cases of application failure using a layered approach to monitoring.

A. Node-Level System Monitoring

Node-level system monitoring often takes the form of a heartbeat mechanism. This is usually maintained by the operating system or coupled with it in the form of a daemon and produces periodic messages to signify that the node is alive. This form of monitoring can be useful in determining if the kernel has crashed, the machine has lost power, or other catastrophic problems have occurred. Linux-HA [1], [2] and HA-Oscar [3], [4] provide this type of solution at the system level.

Monitoring the health of an application at the node-level will catch some sets of problems commonly associated with application failure but it is unable to notice specifics related to the application. The amount of node-level failures compared to other types of failures must be evaluated to determine whether the relative ease of using existing node-level monitoring software is effective in catching a sufficiently large number of problems for a specific target architecture.

B. Subsystem-Level System Monitoring

Modern computing systems are composed of many hardware subsystems and layers system software, middleware and libraries. This convergence of numerous complex systems introduces an unprecedented number of possible failure points. Monitoring the health of individual components provides a better picture of the state of the entire compute resource and would be useful in debugging the points of failure. An automatic heartbeat could be added to existing MPI implementations thereby allowing applications compiled against it to automatically receive this capability.

Some networks, such as InfiniBand, also have a built-in heartbeat mechanism. The Cray Shared File System (SFS)[5] used a HIPPI semaphore (HSMP) device attached to the switch that monitored the heartbeat from active nodes on the HIPPI channel. This was a special-purpose piece of hardware which had other uses but provided this feature and was associated with the shared file system.

C. Application-Level System Monitoring

The ultimate goal is to know whether a specific application is performing useful work. However, determining this can be particularly difficult and usually requires an understanding of the application's internal behavior. Grid-Cop [6] provides a mechanism for developers to instrument their source code with Location Beacons at significant program execution points. These are then monitored to determine if the application is making progress. Source code augmentations are not always possible or desirable. For some applications, source code modifications require re-validation and verification of the scientific results which may be prohibitively expensive. Furthermore, instrumenting a legacy application is often seen as adding additional maintenance costs which may require a long-term investment in the source-level monitoring software.

Daly [7] monitors an application's progress through the periodic growth of program output. This can take the form of textual progress output to the console or growth of a file; such as an output log or data file. This approach benefits from being extremely portable across machines and applications but requires an understanding of the periodic output rate.

Both of these approaches are influenced by the rate at which the application emits status signifying that it is making progress. For the source code augmentation approach the developer may be forced to litter the source with these progress messages which could potentially impact performance and readability of the source. Determining the rate at which an application must reach critical progress points in either case can vary based on the speed of the machine, the node allocation, the compiler optimizations and input parameters among other things. Furthermore, the rate may change in an iterative application during a single run and could result in the expected rate of progress being set conservatively which in turn will result in wasted time when the application hangs and waits to be killed. Generally speaking, for an application which produces a heartbeat at code progress intervals it may be difficult to determine the periodic heartbeat rate as it may vary drastically based on many conditions.

SGI offers a service called Fail Safe [8] where specific applications are monitored remotely through their systems and a central controller is able to determine the current health. Fail Safe is bound to certain application services (such as NFS and HTTPD) but should be easily extendable.

Another approach to application monitoring would be using the kernel to observe the progress of an application by watching the system calls, the use of memory and use of external devices such as the network and disk. It is possible that the kernel could infer the status of the application to determine if progress is in fact being made. This technique however, is the subject of current active research.

III. ANALYTICAL MOTIVATION

For applications that use checkpoint restart as their primary means of fault tolerance, only a fraction of the application's execution time, or *run time* (t_r), is spent performing actual computational work that represents forward progress towards a solution. That time is referred to as *solve time* (t_s). The difference between the solve time and the execution time consists of time spent writing out checkpoint data, restarting after an interrupt, and performing rework to move the calculation from the latest checkpoint back to the point where the interrupt occurred. Daly [9], [10] demonstrated that the relationship between the applications solve time and execution time when checkpointing at regular intervals on a system where interrupts arrive according to a Poisson process can be expressed in terms of the *checkpoint interval* (t_c), *dump time* (δ), *application MTTI* (M), and *restart overhead* (R) as

$$\frac{t_s}{t_r} = e^{-\frac{R}{M}} \left(\frac{\frac{t_c}{M} - \frac{\delta}{M}}{e^{\frac{t_c}{M}} - 1} \right) \quad \text{for } \delta \ll t_s. \quad (1)$$

The dump time is the wall clock time required to create a checkpoint, the application MTTI is the mean time between unscheduled system events that result in an application interrupt, and the restart overhead is wall-clock time elapsed from the point at which the application ceased making progress until the point when it resumes computational work. Daly [9] demonstrated that the ratio of solve time to execution time is maximized when the checkpoint interval is chosen such that

$$t_c \approx \sqrt{2\delta M} \quad \text{for } \delta < \frac{1}{2}M. \quad (2)$$

Much work has been done to understand the impact of the choice of checkpoint intervals on application execution time [11], [12]. Figure 1 demonstrates the interactions of the parameters involved and their relationship to CPdelta. In the ideal case, where CPdelta \ll MTBF, we find a precipitous drop in application efficiency as $\sqrt{2\delta}/M$ increases from 0.1 to 3. The impact of increasing CPdelta is to flatten the entire efficiency curve and drive the maximum achievable application efficiency to zero. Maintaining a ratio of CPdelta to MTBF less than 0.1 keeps the application efficiency in a nominal range.

By choosing the optimum checkpoint interval, the ratio of solve time to execution time actually experienced by the applications will be

$$\frac{t_s}{t_r} \approx \frac{\alpha}{e^\beta} \left(\frac{1 - \frac{1}{2}\alpha}{e^\alpha - 1} \right) \quad \text{where } \alpha = \sqrt{\frac{2\delta}{M}}, \beta = \frac{R}{M}. \quad (3)$$

Equation 3 demonstrates that the ratio of the solve time to execution time is a function of two non-dimensional parameters: α and β . The former is governed by the MTTI of the application and the time required to create the checkpoint file. The latter is additionally governed by the time required to restart the application. Notice also that

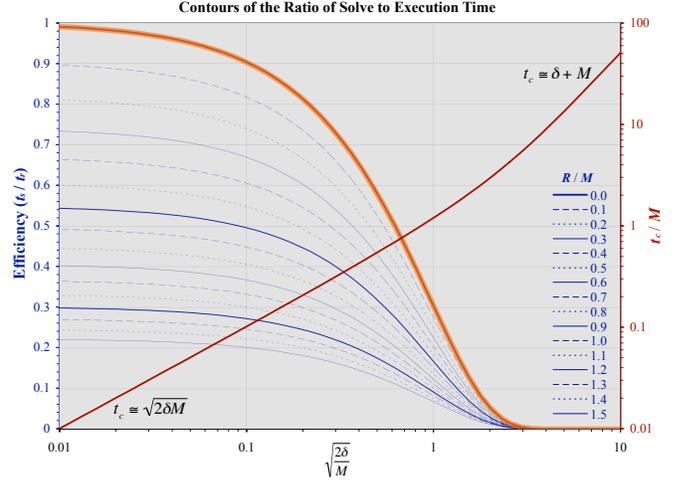


Fig. 1. The runtime efficiency, as a ratio of execution time to solve time, and the optimum time between checkpoints are plotted as functions of the non-dimensional ratio of dump time to MTBF. Contours represent the impact of varying CPdelta relative to MTBF.

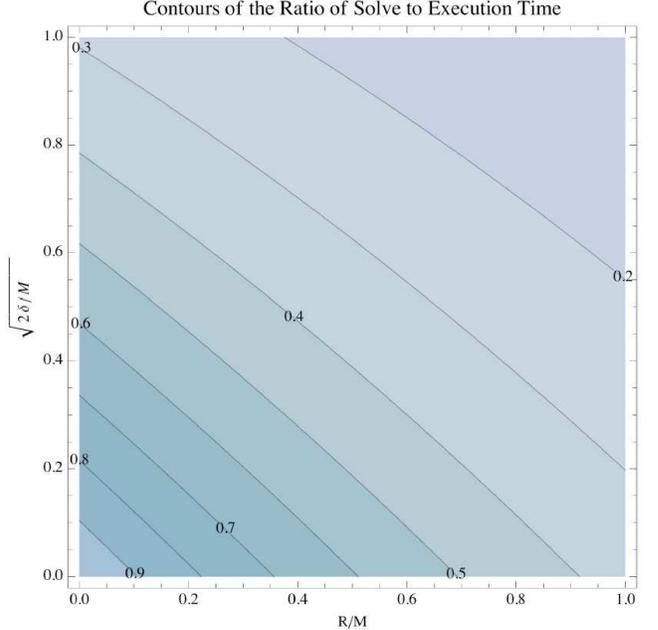


Fig. 2. Results of an analytical model for an application checkpointing at the optimum checkpoint interval between restart dumps. The impact of decreasing CPdelta relative to MTBF can be visualized as shifting a point on the plot to the left, thus increasing the amount of solve time per unit execution time along lines of constant $\sqrt{2\delta}/M$ and improving job turnaround.

when either parameter is small, the exponential terms will be approximately linear. Minimizing α and β has the effect of maximizing the efficiency with which execution time is converted to solve time (i.e., real computational work), as seen in Figure 2. Clearly, the magnitude of the restart overhead will be at least as great as the amount of time required to detect that the application has ceased to make progress. Thus we have the analytical motivation

for minimizing the elapsed time from when the application stops making progress until the time at which it starts making progress again. Furthermore, we see that there is a linear relationship between R and efficiency, particularly in the lower left corner of Figure 2. By decreasing R , we would expect to see a linear increase in efficiency which implies a linear *decrease* in runtime, t_r , for a given MTTI, M .

IV. SIMULATION

In this section, we describe the simulation-based study conducted to illustrate the impact of several parameters on application resilience. In particular, we’re interested in how quickly the application monitoring subsystem must make the determination that an application is no longer making progress. As such, we conducted a parameter study across the CPdelta and MTBF dimensions. In this context, CPdelta refers to the amount of time that elapses between when an interrupt occurs and the application monitor realizes that it has occurred i.e. the detection latency. This is an important component of the overall restart overhead, R , as seen in Equation 3. Based on the various approaches discussed above, we’ve elected to sweep CPdelta from around 1 minute to a maximum of 46 minutes. This represents a reasonable range based on existing application monitoring techniques. In addition to the CPdelta dimension, we’ve considered MTBFs ranging from 3 to 24 hours in order to illustrate the impact of increased failure rates. Note, we consider a failure to be anything that causes an application to stop making progress, including transient hardware errors, system software interruptions, as well as application bugs.

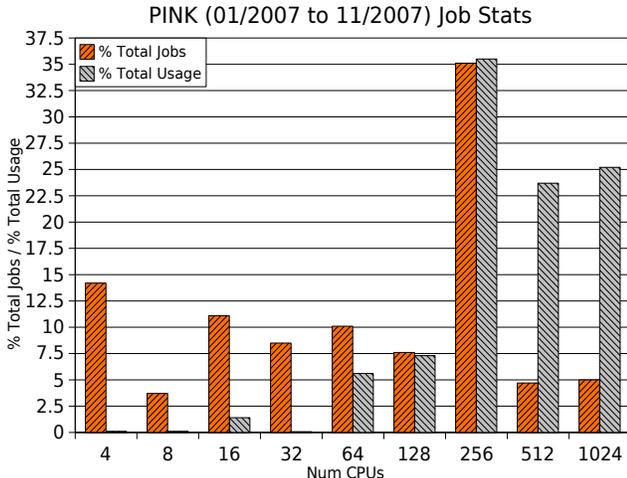


Fig. 3. Job size and usage distributions used in the simulation. These are based on a 1-year trace of jobs on LANL’s PINK cluster. *Note that although there are significant numbers of smaller jobs, they do not constitute a large fraction of the total usage.*

Our cluster simulator [13], [14] is parametrized to model the Pink cluster at Los Alamos National Laboratory

(LANL). Specifically, Pink is a 1926 node cluster available to LANL collaborators and is generally used for computational science. For the initial experiments, our simulator’s workload generator has been tuned to both the sizes and runtimes of jobs typically seen on Pink. This was accomplished using a 1-year statistical summary of Pink’s management system as seen in Figure 3. We assume that jobs arrive according to a Poisson process, and have adjusted the interarrival times to load the system. Furthermore, we have assumed that the TBFs are also exponentially distributed. In our simulator, as a failure “arrives”, we select a system node at random as the originator of the failure, and then mark any job mapped to the chosen node as failed. We then expose this failure to the application monitor after a prescribed delay, $CPdelta$, to evaluate the impact that various latencies have on application execution time and overall throughput.

While this approach establishes a first step towards evaluating the impact of failures on system performance, a more sophisticated scheme would distinguish among various failure types that directly effect multiple jobs. Additionally, note that a Poisson failure process may not closely approximate a realistic failure distribution. While this makes the analysis more tractable, further investigation using a Weibull distribution, for example, is warranted.

V. RESULTS AND OBSERVATIONS

In Figure 4, we see that the average application turnaround time is fairly linear with respect to CPdelta, especially for the larger MTBFs (lower failure rates). This is consistent with the analytical model described in Section III. While it is intuitive that reducing CPdelta would lead to better overall system performance, it is important to note that eliminating CPdelta altogether is not necessary. A reduction to around 5 minutes is sufficient. This is important because near-instantaneous fault detection can be expensive, may limit portability, and may not be pragmatic in many situations due to security or potential impact to the code-base.

Since application turnaround time (queue waiting time + execution time) is a function of system load, it is important to also consider metrics that are load invariant. In this context a job’s execution time is effected by CPdelta as well as the checkpoint overhead. For example, when a job experiences an interrupt, it will loose all work that has been completed since the last checkpoint (CP) and will also later incur the the overhead of a restart. Interrupted jobs will therefore experience a degradation in their execution-time performance. Figure 5 shows the execution time averaged across interrupted jobs. Reducing CPdelta from 46 minutes to 6 minutes, a 12% improvement is obtained in execution time. Moreover, the additional benefit obtained from near-immediate failure detection is minimal. This suggests that a more practical latency of 5 or 6 minutes may be a sufficient requirement for the

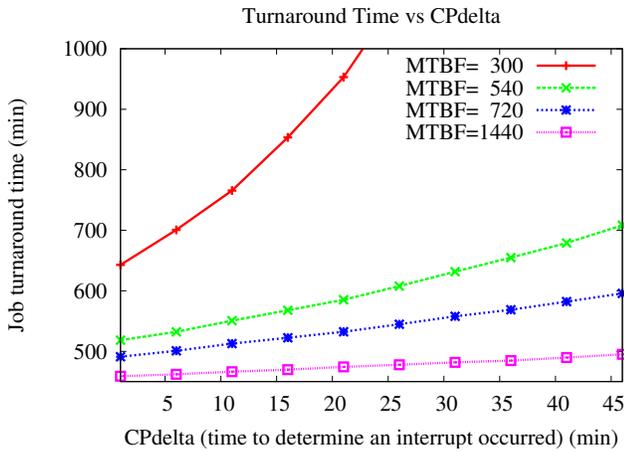


Fig. 4. Turnaround time as a function of CPdelta at various failure rates. Note that as failures become more frequent, the overall performance becomes more sensitive to reductions in the detection latency, CPdelta.

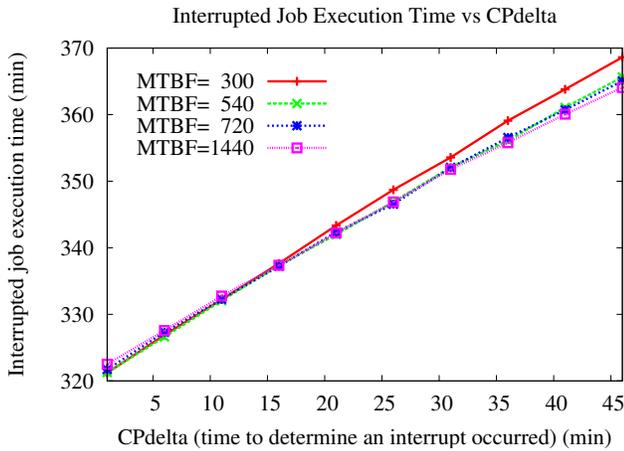


Fig. 5. Job runtime as a function of CPdelta. Note that there is a significant benefit to decreased CPdelta for interrupted jobs. In this case, around 12% reduction in apparent execution time by decreasing CPdelta from 46 min to 6 min. Furthermore, note that minimal improvement is obtained by decreasing the detection latency to 0.

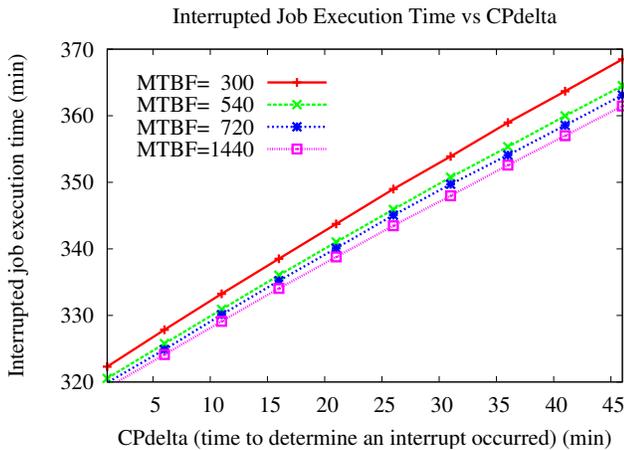


Fig. 6. Job runtime as a function of CPdelta. Here the initial job runtime distribution is the same for all classes of jobs. This is done to illustrate the expected relationship between MTBF and execution time due to multiple single-application interruptions. Note the similar improvement due to a reduction in CPdelta.

application monitoring subsystem. Since only jobs that are interrupted see a degradation in execution time, it naturally follows that the extent to which the overall average execution time is improved would depend not only on the improvement seen by the interrupted jobs, but also on the overall number of jobs that experience an interrupt. As the MTBFs decreases, the fraction of jobs that experience a failure increases. Furthermore, an application may experience more than one interrupt, thereby increasing its apparent execution time.

One interesting artifact in Figure 5, is that there does not appear to be much impact on execution time as a function on decreasing MTBFs. This may be counter-intuitive; however, it should be noted that the workload contains a correlation between the width of the jobs and the corresponding initial runtime distributions as seen in Figure 3. As the failure rate increases, a larger fraction of the shorter-running jobs are interrupted. As such, the execution-time average is composed of a larger number of initially shorter running jobs. This balances the degradation seen by longer running jobs as a result of multiple interruptions. To clarify this point, we conducted a similar study where the runtime distribution is the same for all classes of jobs. In Figure 5 we see the expected trend described above.

It is also worth noting that even at the higher failure rates, the total percentage of interrupted jobs is fairly low, around 6.5%, as seen in Figure 7. This may seem unusual; however recall that large longer-running jobs, which make up a large percentage of the workload mixture (Figure 3), are particularly vulnerable to multiple interruptions.

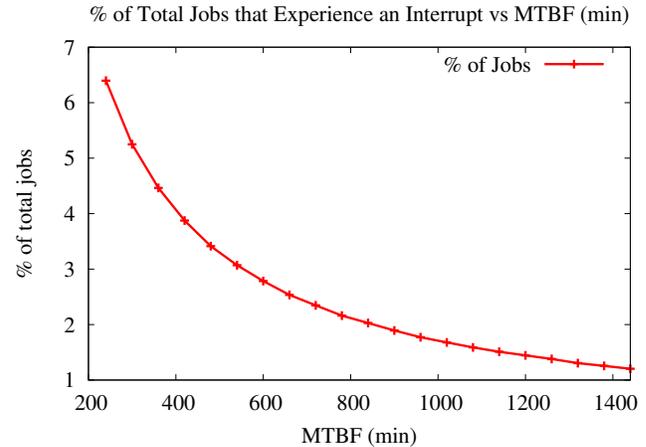


Fig. 7. Percentage of total jobs (out of 4,000,000 per simulation run) that experience an interrupt as a function of MTBF. Note that while the percentage may seem unexpectedly low at the higher failure rates, jobs may experience multiple failures between restarts. This is particularly the case for longer-running jobs.

While these observations suggest that there are minor gains to reducing the detection latency below around five minutes, there are systems where further reduction would be beneficial. For instance, transaction and real time

data processing systems usually have limited tolerance for errors and associated latencies. Additionally, it should be noted that the appropriate detection latency may be application dependent or correlated to the failure type. In our simulations, node failures occur according to a Poisson process; however other failures, such as wide-spread rack or switch failures that impact multiple jobs need to be characterized and studied in this context. As a first step, we have assumed CPdelta to be independent of the interrupted job's characteristics and conduct a parameter sweep across varying CPdeltas to see its correlation to execution time and application throughput.

VI. CONCLUSIONS

While it is important to make every effort to avoid failures whenever possible and cost effective, it is of equal importance to recognize that failures will inevitably occur and to continue to make progress in spite of these failures. Detecting an application's failure to make forward progress is particularly important in large cluster systems. Once an application fails, it may continue to consume valuable node resources that other queued jobs could take advantage of. Additionally, detecting such failures more quickly will reduce a jobs apparent execution time.

In this paper, we have suggested several potential application monitoring frameworks with a wide range of complexities as well as detection latencies. Some of these techniques are complex and expensive while others are novel and portable. Some require specialized hardware, kernel modifications, specialized libraries or changes to the target application. They each result in different detection latencies.

We have conducted a simulation-based study that indicates that near-instantaneous failure detection is not required to achieve the majority of potential improvement in execution-time performance. These conclusions are also supported by the associated analytical models.

This conclusion is significant because it implies that less complex, highly portable, and predominately less expensive failure detection schemes would provide adequate application resilience.

VII. FUTURE WORK

After detecting that a failure has occurred, the application monitor must communicate with the scheduling framework in order to determine how to proceed. In this paper, we have assumed that the scheduling policy requires that the interrupted job be restarted as soon as possible; however this need not be the case. For example, suppose that historical information is used to identify jobs that are predisposed to fail. In this case, the scheduler could make a more informed decision to ensure fairness in the presence of buggy application code.

As previously mentioned, a more sophisticated model of failure processes would help distinguish among failures

that directly impact multiple jobs. For example, multiple jobs might be effected by a single switch failure. An analysis of failure types and their associated impacts would provide insight into more precisely determining the optimal checkpoint interval based on the failure characteristics of the resources it is mapped across.

Additionally, addressing "soft failures", i.e. failures that simply degrade performance as opposed to those that catastrophically halt progress, would allow an application monitor to determine whether dynamically reallocating resources based on conditions would in fact lead to improved performance. This is of particular importance in addressing intelligent parallel job scheduling strategies in computational grids where many jobs can be migrated [15] and are moldable.

REFERENCES

- [1] A. Robertson, "Linux-ha heartbeat design," in *Proceedings of the 4th International Linux Showcase and Conference*, 2000.
- [2] —, "The evolution of the linux-ha project," in *UKUUG LISA/Winter Conference High-Availability and Reliability*, 2004.
- [3] C. Leangsuksun, L. Shen, T. Liu, H. Song, and S. Scott, "Availability prediction and modeling of high availability oscar cluster," *Cluster Computing*, 2003. *Proceedings. 2003 IEEE International Conference on*, pp. 380–386, 1-4 Dec. 2003.
- [4] J. Mugler, "Oscar clusters," in *John Mugler, et al. OSCAR Clusters, Proceedings of the Ottawa Linux Symposium (OLS'03), Ottawa, Canada, July 23-26, 2003.*, 2003.
- [5] K. Matthews, "Implementing a shared file system on a hippi disk array," *Mass Storage Systems, 1995. 'Storage - At the Forefront of Information Infrastructures', Proceedings of the Fourteenth IEEE Symposium on*, pp. 77–88, 11-14 Sep 1995.
- [6] S. Yang, A. R. Butt, Y. C. Hu, and S. P. Midkiff, "Trust but verify: monitoring remotely executing programs for progress and correctness," in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2005, pp. 196–205.
- [7] J. T. Daly, "Facilitating high-throughput asc calculations," in *Nuclear Weapons Highlights*. Los Alamos National Laboratory (LALP-07-041), 2007, pp. 202–203.
- [8] "IRIS FailSafe – The SGI High-Availability Solution Website," <http://www.sgi.com/products/software/failsafe/overview.html>.
- [9] J. T. Daly, "A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps," *Future Generation Computer Systems*, vol. 22, pp. 300–312, 2006.
- [10] —, "Methodology and metrics for quantifying application throughput," in *Proceedings of the Nuclear Explosives Code Developers Conference*, 2006.
- [11] B. Schroeder and G. Gibson, "Understanding failures in petascale computers," *J. Phys.: Conf. Ser.*, 2007.
- [12] X. S. M. Wu and H. Jin, "Performance under failures of high-end computing," in *Proc. ACM/IEEE Super Computing Conf.*, November 2007.
- [13] W. M. Jones, L. W. Pang, D. Stanzione, and W. B. Ligon III, "Characterization of bandwidth-aware meta-schedulers for co-allocating jobs across multiple clusters," in *Journal of Supercomputing, Special Issue on the Evaluation of Grid and Cluster Computing Systems*, vol. 34, no. 2. Springer Science and Business Media B.V, November 2005, pp. 135–163.
- [14] "BeoSim Website," <http://www.parl.clemson.edu/beosim>.
- [15] W. M. Jones, "Using checkpointing to recover from poor multi-site parallel job scheduling decisions," in *The 5th International Workshop on Middleware for Grid Computing (MGC 2007), held in conjunction with, ACM/IFIP/USENIX 8th International Middleware Conference 2007 (Middleware 2007)*, November 2007.