Simulating Failures on Large-Scale Systems

Narayan Desai[#], Ewing Lusk[#], Daniel Buettner[†],
Andrew Cherry[†], Theron Voran[*]

[#]*Mathematics and Computer Science Division*

[†]*Leadership Computing Facility*
Argonne National Laboratory, Argonne, IL 60439, USA
{desai,lusk}@mcs.anl.gov
{buettner,acherry}@alcf.anl.gov

[*]*Computer Science Department, University of Colorado*
Boulder, CO 80309, USA
theron.voran@colorado.edu

***Abstract*—Developing fault management mechanisms is a difficult task because of the unpredictable nature of failures. In this paper, we present a fault simulation framework for Blue Gene/P systems implemented as a part of the Cobalt resource manager. The primary goal of this framework is to support system software development. We also present a hardware diagnostic system that we have implemented using this framework.**

I. Introduction

The scale of high-end computing (HEC) systems has continued to grow over the past twenty years. The largest system on the most recent Top500 [1] list has over 200,000 cores; and the next version of this list will likely contain several systems with more than 50,000 cores. Fault management is a key issue on these systems and is of growing importance on medium scale systems in wider use.

These HEC systems contain enormous numbers of components that are used in parallel. This parallel workload has important implications for fault sensitivity: in many cases, a single component hardware or software failure can result in an overall job or system failure involving hundreds or thousands of components.

Component mean-time-between-failure rates combine catastrophically with component counts; the mean time between interrupt for large-scale or full system jobs can be drastically low. Moreover, even when a failure occurs and is detected, the culprit may not be known.

Developing fault tolerance techniques for such large-scale systems can be difficult. Component hardware failures can be hard or impossible to trigger. In addition, some fault behaviors result from complex chains of failures; these behaviors are impossible, in practice, to trigger.

In this paper, we describe the fault simulation framework we have implemented in Cobalt [2], a popular resource manager on the IBM Blue Gene/L and Blue Gene/P platforms. This framework provides a convenient setting for fault tolerance development. We also describe a simple fault diagnostic system that we developed using this framework. We conclude with some observations about the key features of this framework and present our path forward.

II. Background

In this section, we describe the background for the remainder of the paper. We begin by describing the system setting for our work, the IBM Blue Gene. From there, we present a more concrete and detailed discussion of failures in HEC systems. We conclude the section with a discussion of Cobalt, a system software suite that we have developed.

A. Intrepid

Intrepid, a 40-rack BG/P system with peak performance of 556 TF, was deployed at Argonne last fall. This system, comprised of 163,840 cores, is operated as part of the U.S. Department of Energy effort to provide leadership-class computing resources to the scientific community. DOE selects major projects through the INCITE (Innovative and Novel Computational Impact on Theory and Experiment) [3] program to run a capability workload. While large parallel systems usually see a large number of small- to medium-sized jobs, Intrepid has the opposite workload: a small number of projects are given substantial allocations. These projects have been screened for computational readiness; each is ready to run at large scale. Often, all resources on Intrepid are consumed by fewer than a dozen jobs.

The IBM Blue Gene family of systems is a series of integrated MPP-style systems. Each system consists of a series of racks, each containing 1,024 multicore nodes. Blue Gene/P, the most recent generation of these systems, uses a four-core processor, for a total of 4,096 cores, or 13.9 TF, per rack. The 72-rack maximum configuration of this system provides just over 1 PF of peak performance. We will not discuss the system architecture in much detail; more information is available elsewhere [4].

Blue Gene systems have several unusual characteristics compared with traditional cluster-style systems. Nodes are grouped into partitions, where jobs can be executed. Several restrictions govern how partitions can be constructed. Nodes are grouped into *psets*. Each pset consists of a single I/O node, responsible for external connectivity and persistent I/O, and a group of compute nodes. The size of this node group, called pset size, ranges from 16 to 128; Intrepid has a pset size of 64. Psets are grouped into midplanes. Midplanes always contain 512 nodes and provide the smallest functional torus network for application communication. Instead of the torus network, partitions smaller than one midplane have access only to a mesh network with identical link performance properties. Multi-midplane partitions can be created only if sufficient network links exist to connect individual midplane torus networks into a larger torus or mesh network.

Another unique aspect of Blue Gene systems is the introduction of a control system. This software is responsible for all system management functions, including hardware maintenance, job execution, and system monitoring. While these functions exist on a traditional system, they are typically not implemented in a unified and scalable fashion.

B. Failure Diagnosis

No components, hardware or software, used in HEC systems are universally reliable. Hardware failures are unavoidable, particularly as the scale of systems grows. On a system of Intrepid's scale, failure management is a substantial concern.

When these failures occur, the source and nature of the problem must be determined. The best-case scenario for diagnosis is outright component failure. In many cases, however, marginal components perform poorly or fail irregularly. The marginal case is typically much more difficult to diagnose.

On parallel systems, diagnosis is complicated by the system workload. The system infrastructure used by large-scale systems frequently makes use of aggregation. On Blue Gene systems, individual compute nodes are grouped into partitions, where jobs can be run. When a fault occurs during job execution, the exact location of the problem may not be immediately known. Some failures are reported with a precise location; however, many others are not. This problem is particularly relevant on capacity systems, where job sizes are frequently a large fraction of the system size.

Blue Gene systems ship with a set of hardware diagnostics. These diagnostics are comprehensive and high quality but must be run offline. The diagnostics can be run frequently, but at the cost of system availability.

C. Cobalt

Cobalt [2] is a resource management suite that has grown out of the Scalable System Software SciDAC project [5]. Cobalt is designed around a component architecture. Each major piece of functionality is a discrete component that operates as a independently. Components communicate bby using XML-RPC secured with SSL. Cobalt is relatively small, comprising less than 12,000 lines of python code.

Component architectures provide several benefits. Generally, component systems reduce duplication, as a single implementation of functionality can be used from all potential consumers. The use of a well-defined component interface decouples providers of functionality from consumers. This abstraction is beneficial in both directions. New consumers can be added without modifying providers, and provider implementations can be replaced without modifying consumers. Each of these characteristics has proved useful as we have developed and deployed Cobalt.

Cobalt is used primarily on IBM Blue Gene/L and Blue Gene/P systems, but it was originally developed and used on commodity Linux clusters. Its component architecture made the port to Blue Gene/L systems quite easy. This process is detailed in [6]. It has now been deployed on more than a dozen Blue Gene/L and Blue Gene/P systems worldwide. Intrepid is the largest and most prominent of these systems.

Beginning in summer 2006, Cobalt development efforts were stymied by the lack of dedicated development resources. At that time, Cobalt was run on a single-rack BG/L at Argonne. While the system's mission was a combination of computational and computer science, dedicated access to the entire system – as needed for resource manager development – was hard to come by.

To address this issue, we developed a simulator that mimicked system behavior, including both proper behavior and failures caused by resource manager failure [7]. In effect, this simulation replicated the expected system behavior. The component infrastructure of Cobalt allowed us to run a development configuration using simulation without changing the balance of the Cobalt code. In addition, we were even able to switch between native and simulated behavior on the fly. Our work presented here is an extension of that earlier work.

D. Simulation

Simulation is frequently used in software and system design. System designers often use simulators to bring up initiatl software before real hardware is available. For example, system simulation was used for initial work on Blue Gene/L systems.

Simulators also are a mainstay in fault tolerance research becuase of the difficulty in triggering faults [8], [9]. Moreover, simulators have found widespread use in performance prediction, including performance studies of large-scape systems [10], [11].

Simulation also has found some use in HPC system software, most prominently in the Maui scheduler [12]. In this case, simulation provides the ability to predict the effects of a scheduling policy on a given workload. This simulation is primarily inward facing; that is, it provides insight into the behavior of only a single system software component, as opposed to the entire system software ecosystem.

III. Approach

The system simulation capabilities in Cobalt were designed with several use cases in mind. Our initial target was the set of Cobalt developers themselves. When we began to have trouble getting dedicated access to our single-rack Blue Gene/L, we began to use Cobalt to test new software releases in advance of deployment. When we transitioned to Intrepid, fault management became much more pressing. At this point, we added fault injection capabilities for individual components. As we gained more experience operating this system, it became obvious that hardware fault management was an issue that needed to be directly addressed. This prompted us to begin work on a hardware diagnostic system.

In this section, we describe our approach in detail, highlighting important design criteria and operational issues. First, we discuss the design and implementation of Cobalt's system simulator component, including the fault injection constructs. Next, we describe the hardware diagnosis mechanism we have implemented.

A. System Simulator

Cobalt implements system simulation through the use of component implementation replacement. All interactions with the underlying hardware, including job execution, partition monitoring, and hardware management, are abstracted behind the interface to a single component, the *system* component.

Most simulators, particularly those used for fault tolerance research, mimic the function of low-level processes. The simulator implemented in Cobalt takes the opposite approach, implementing high-level system behavior. On real hardware, this is the synthesis of individual component behaviors into high-level systemic behavior.

This approach is particularly useful for system managers. Often, problems will occur intermittently, and in some cases it may be impossible to determine root causes. The lack of specificity used in Cobalt's simulator allows the implementation of observed system behaviors, even if the exact cause is not known.

1) Implementation: Two implementations of the system component exist. The first is a native implementation that communicates directly with the Blue Gene control system supplied by IBM. This implementation queries the control system to track partition states, hardware status, and user jobs.

The second system component implementation is a simulator. This component is provided with a system definition upon startup. This system definition describes the size of the system, all configured hardware components, and workable combinations of these components into partitions. `bgsystem` can generate these definitions from an actual Blue Gene system upon demand, during normal execution.

The two system component implementations share a substantial fraction of their code. `bgsystem` implements only those functions needed to probe the control system for current state and launch new jobs. The simulator likewise implements internal state management and job execution simulation.

In simulation, job execution is performed by mimicking the output that a user would see in terms of control messages, and sleeping for a random percentage of the requested wall-clock time. Figure 1 shows an example of simulated failure output.

```
FE_MPI (Info) : Initializing MPIRUN
BE_MPI (ERROR): Booting aborted - partition is in DEALLOCATING ('D') state
BE_MPI (ERROR): Partition has not reached the READY ('I') state
BE_MPI (Info) : Checking for block error text:
BE_MPI (ERROR): block error text 'side fumbling detected.'
BE_MPI (Info) : Starting cleanup sequence
BE_MPI (Info) : Partition ANL-R00-R01-2048 switched to state FREE ('F')
FE_MPI (ERROR): Failure list:
FE_MPI (ERROR): - 1. ANL-R00-R01-2048 couldn't boot.
FE_MPI (Info) : FE completed
FE_MPI (Info) : Exit status: 1
```

Fig. 1.   Simulated Failure Output

A key behavior of the simulator is failing under circumstances where a Blue Gene system would also fail. For example, each node can participate only in a single partition at a time, and each partition can run only a single job at once. If two jobs are run on a Blue Gene system on either the same partition or overlapping partitions, the first will succeed and the second will fail.

2) Fault Simulation: The system simulator includes an interface for fault injection. The simulator is provided with a complete system description upon startup; this description includes a comprehensive component list. Each component can be set to failing status. In this mode, any action on a partition including the failing resource fails. In the case of job execution, a job will sleep for a random period of time less than wall clock time and then fail.

The simulator deals properly with single and multiple failures. All failures are treated as consistent failures; multiple actions on the same failing simulated resources will fail in a similar fashion.

B. Fault Isolation

The fault isolation process locates failing hardware components based on observed behavior. On a large-scale system like Intrepid, observed behavior frequently integrates the behavior of massive quantities of components into a single result. Low-level diagnostics can also be useful but scale and perform poorly. Each of these approaches is lacking: the first in terms of detail and the second in performance. Clearly, a hybrid approach is needed.

1) Fault Detection: Tests are either fast or detailed, but rarely both. The diagnostic routines shipped with Blue Gene systems are specific but run slowly. While these diagnostics are effective at locating failing hardware, they do not scale well. Execution of diagnostics on a single-rack partition takes approximately one hour, and the system is capable of executing only two of these tasks simultaneously. Moreover, during parallel executions of diagnostics, substantial slowdowns occur. Intrepid is a capability system; its average job size involves more than two racks, and the largest job sizes can require considerably more. While diagnostics are a useful tool for locating failing hardware, these factors all combine to prevent them from being a comprehensive solution. If a failure is detected on an 8-rack partition, serialized diagnostics could easily take 8 hours to perform.

Other tests run quickly but are less detailed. Some user applications provide a sweet spot between speed and detail. Many of these applications perform substantial internal checking to detect problems during execution. When an inconsistency is detected, the application detects the problem and halts. These checks can be performed fairly quickly but only verify the presence of a problem somewhere in the partition where the application was run. The obvious path to faster fault isolation is to use fast tests to localize problems. Once these have been done, detailed diagnostics isolate the exact location of the problem, so that hardware maintenance can be performed.

2) Fault Localization: The heart of this approach is to use general (and fast) diagnostics to reduce the need for time-consuming complete diagnostics. The most common hardware failures result in persistent behavior, so a simple execution of a test (or set of tests) is sufficient to determine the proper function of a set of hardware components. If a test fails to execute properly on a partition, then the partition is subdivided and the test is executed on smaller groups of components. When partitions pass all tests, they are returned to service.

This approach depends on the job execution mechanism for Blue Gene systems. It is capable of running large numbers of simultaneous jobs. Hence, user jobs cause much less infrastructure load than do comparable diagnostics.

3) Implementation: Both system component implementations, `bgsystem` and the simulator, can use the same test orchestration code. The only difference between the implementations of diagnostics is the literal test spawning code. The simulator, as expected, simulates the results of tests based on the current fault status of pertinent components. `bgsystem` spawns tests and determines their success based on their exit statuses.

The diagnosis process is initiated upon administrator request. This request consists of a suspected faulty partition and a test mode. The first action taken is to set all resources included in the partition offline, so that no subsequent user jobs will be executed. If the partition is currently idle, testing commences immediately; otherwise the process is delayed until current jobs conclude. Figure 2 shows the system with several diagnostics running.

```
# ./partadm.py -l
Name            Queue    Size  Funct  Sched  State
==============================================================
R00-R01-2048    default  2048    X      X    blocked (R01-M0-512)
R00-1024        default  1024    X      X    diags
R01-1024        default  1024    X      X    blocked (R01-M0-512)
R00-M0-512      default  512     X      X    idle
R00-M1-512      default  512     X      X    idle
R01-M0-512      default  512     X      X    busy
R01-M1-512      default  512     X      X    diags
```

Fig. 2. Partition State during Diagnostics

If a specific test is requested, that test is executed on the partition. If no test is specified, each of the short, application-based tests is run because each finds different types of issues. If these tests pass, all resources are returned to service.

Each test consists of a script that Cobalt executes; the exit status conveys the result of the test. Because systems encounter different problems over time, owing to aging of hardware and changes to software, an extensible testing facility was needed.

In the event of a test failure, the failing partition is subdivided into a covering set of smaller partitions. The same test is run in parallel on each of these. If possible, a binary search is performed; but because of the flexibility of system partitioning, this is not always possible. We use this approach because our application-based tests run a single problem size regardless of partition size. For this reason, the fast tests run considerably faster on a large partition than on a small partition.

Once subdivision results in an individual failing midplane, full-scale diagnostics are executed. By this point, all properly functioning resources have been returned to service. System diagnostics determine the exact location of the hardware fault, so that hardware replacement can be performed.

IV. Results

The implementation of the simulator itself is straightforward. Moreover, it enabled the development of failure management software in a standard fashion.

Dedicated time on a system such as Intrepid is difficult to obtain. Development time during system failures is even more problematic, since failures cannot be easily predicted. The open testing architecture described here has worked well, enabling us to run a range of tests to detect failures.

V. Conclusions

Fault management is an increasingly important task on large-scale HPC systems. In this paper, we have presented a fault injection mechanism that we have implemented in the Cobalt resource manager. We have demonstrated the utility of this approach in the design and implementation of a failure search mechanism for the Intrepid system at Argonne. From these experiences we can draw several conclusions.

Simulation is the only realistic approach for conducting work in large-scale fault tolerance. The difficulty of triggering individual faults, in conjunction with the need to orchestrate them similarly to their occurrence in large-scale systems, makes development in a natural setting impractical.

The combination of simulation with component architectures is quite powerful. The vast majority of the code needed to implement the diagnostic system can be developed and tested without access to actual hardware, thanks to the system simulation component. Because the two system component implementations share much of their code, results on one instance are largely representative of result on the other. This infrastructure made it easy to develop the diagnostic system.

The implementation of the diagnostic system has also provided interesting insights. Low-level diagnostics are much slower than high-level application testing that synthesize many aspects of system operations. The cost of running high-level diagnostics of this style is that failures are not typically localized, so a search process must be performed. As system scales continue to grow, we expect this issue to occur more frequently.

A. Future Work

This work provides a solid foundation to build , both in terms of a framework to explore fault management, as well as a diagnostic system. The fault injection mechanism supports the simulation of only single component failures. As system scales increase, as does the likelihood of hardware double-faults. It remains to be seen whether more effective mechanisms can be designed to detect and locate this eventuality. The system currently simulates only persistent failures; yet particularly in the case of marginal hardware, intermittent failures are common. Building a model for these failures would be a useful addition.

We have implemented in the system simulator a class of hardware failures behaviors that correspond to consistent faults. As we experience other persistent fault behaviors, we will implement them also.

The simulator is currently ignorant of time. We plan to add time sensitivity; this will allow us to specify a series of faults over time whose behavior can be simulated appropriately.

Cobalt has been integrated with the CIFTS fault-tolerant backplane (FTB) [13], which provides scalable communication of fault events between system software components. The fault event information provided by the FTB will allow for more nuanced and accurate simulation of experienced system faults.

This work provides a basic mechanism for automated hardware diagnosis, but improvement is still needed. The current mechanism must be manually triggered, in effect using the system administrator as an expert system. We need to begin implementing recognition strategies that can either trigger diagnostics directly or cause low priority diagnostics to be executed as scheduling backfill.

As we mentioned in Section III-B1, system managers are faced with a difficult choice of diagnostic frequency. Clearly, different approaches are needed on different systems and with different workloads. Quantitative studies, analogous to those performed for checkpointing [14], are needed to provide guidance.

Our current tests do not choose a problem size based on the scale of partition diagnosed. This approach might allow us to run fast diagnostics in parallel, should this result in better performance.

Availability

The fault injection capability has been implemented in the Cobalt codebase and will be included in the version 0.98.2 release. Cobalt is BSD licensed and publicly available. Software, documentation, and papers are available from the Cobalt web site [2].

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. Other support was from the DOE SciDAC program under award #DE-FG02-04ER63870, NSF MRI Grant #CNS-0420873, and the NSF sponsorship of the National Center for Atmospheric Research.

References

[1] Top500 Web page. Top500.Org. [Online]. Available: http://www.top500.org

[2] N. Desai. Cobalt Web page. Argonne National Laboratory. [Online]. Available: http://trac.mcs.anl.gov/projects/cobalt

[3] Innovative and novel computational impact on theory and experiment (incite) program. US Department of Energy. [Online]. Available: http://www.er.doe.gov/ascr/incite/index.html

[4] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, "Overview of the blue gene/l system architecture," *IBM Journal of Research and Development*, vol. 49, no. 2-3, pp. 195–212, 2005.

[5] A. Geist. SciDAC scalable system software Web page. Oak Ridge National Laboratory. [Online]. Available: http://www.scidac.org/ScalableSystems

[6] N. Desai, R. Bradshaw, A. Lusk, E. Lusk, and R. Butler, "Component-based cluster systems software architecture: A case study," in *Proceedings of the 6th IEEE International Conference on Cluster Computing (CLUSTER04)*. IEEE Computer Society, 2004, pp. 319–326.

[7] N. Desai, E. Lusk, A. Cherry, and T. Voran, "The computer as software component: A mechanism for developing and testing resource management software," in *Proceedings of the 9th IEEE International Conference on Cluster Computing (CLUSTER07)*. IEEE Computer Society, 2007, pp. 58–63.

[8] I. Koren. Fault tolerant systems simulator Web page. University of Massachusetts, Amherst. [Online]. Available: http://www.ecs.umass.edu/ece/koren/FaultTolerantSystems/simulator/

[9] S. Das, P. Flocchini, N. Santoro, and M. Yamashita, "Fault-tolerant simulation of message-passing algorithms by mobile agents," in *SIROCCO*, ser. Lecture Notes in Computer Science, G. Prencipe and S. Zaks, Eds., vol. 4474. Springer, 2007, pp. 289–303.

[10] G. Zheng, G. Kakulapati, and L. V. Kalé, "Bigsim: A parallel simulator for performance prediction of extremely large parallel machines," in *IPDPS*. IEEE Computer Society, 2004.

[11] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalé, "Simulation-based performance prediction for large parallel machines," *Int. J. Parallel Program.*, vol. 33, no. 2, pp. 183–207, 2005.

[12] Maui Web page. Cluster Resources, Inc. [Online]. Available: http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php

[13] P. Beckman. CIFTS Web page. Argonne National Laboratory. [Online]. Available: http://www.mcs.anl.gov/research/cifts

[14] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. S. Gupta, "Performance implications of periodic checkpointing on large-scale cluster systems," in *IPDPS*. IEEE Computer Society, 2005.