Available online at www.sciencedirect.com
SCIENCE DIRECT®
Future Generation Computer Systems 22 (2006) 293–302
FGCS
FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/fgcs

Reliability challenges in large systems

Daniel A. Reed[a,*], Charng-da Lu[b], Celso L. Mendes[b]

[a] *Renaissance Computing Institute, University of North Carolina, Chapel Hill, 27599 NC, USA*
[b] *Department of Computer Science, University of Illinois, Urbana, 61801 IL, USA*

Available online 1 January 2005

Abstract

Clusters built from commodity PCs dominate high-performance computing today, with systems containing thousands of processors now being deployed. As node counts for multi-teraflop systems grow to tens of thousands, with proposed petaflop system likely to contain hundreds of thousands of nodes, the assumption of fully reliable hardware and software becomes much less credible. In this paper, after presenting examples and experimental data that quantify the reliability of current systems, we describe possible approaches for effective system use. In particular, we present techniques for detecting imminent failures in the environment and that allow an application to run successfully despite such failures. We also show how intelligent and adaptive software can lead to failure resilience and efficient system usage.

Keywords: System reliability; Fault-tolerance; Adaptive software

1. Introduction

Clusters built from commodity PCs dominate high-performance computing today, with systems containing thousands of processors now being deployed. As an example, the National Center for Supercomputing Applications (NCSA) has deployed a 17.9 teraflop Linux cluster containing 2938 Intel Xeon processors. Even larger clustered systems are being designed and deployed – the 40 teraflop Cray/SNL Red Storm cluster and the 180 teraflop IBM/LLNL Blue Gene/L system will contain 10,368 and 65,536 nodes, respectively. As node counts for multi-teraflop systems grow to tens of thousands, with proposed petaflop systems likely to contain even larger numbers of nodes, the assumption of fully reliable hardware and software becomes much less credible.

Although the mean time before failure (MTBF) for the individual components (i.e., processors, disks, memories, power supplies, fans and networks) is high, the large overall component count means the system itself may fail frequently. For example, a system containing 10,000 nodes, each with a mean time to failure of 10^6 h, has a system mean time to failure of only 100 h, under the generous assumption of failure independence. Hence, the mean time before system failure

* Corresponding author.
E-mail address: dan_reed@unc.edu (D.A. Reed); clu2@cs.uiuc.edu (C. Lu); cmendes@cs.uiuc.edu (C.L. Mendes).

0167-739X/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2004.11.015

for today's 10–20 teraflop commodity systems can be only 10–40 h, due to a combination of hardware component and software failures. Indeed, operation of the large IBM/LLNL ASCI White system has revealed that its MTBF is only slightly more than 40 h, despite continuing improvements.

For scientific applications, MPI is the most popular parallel programming model. However, the MPI standard does not specify mechanisms or interfaces for fault-tolerance – normally, all of an MPI application's tasks are killed when any of the underlying nodes fails or becomes inaccessible. Given the standard domain decompositions and data distributions used in message-based parallel programs, there are few alternatives to this approach without underlying support for recovery. Intuitively, because the program's data set is partitioned across the nodes, any node loss implies irrecoverable data loss.

Several research efforts have attempted to address aspects of this domain decomposition limitation. Historically, large-scale scientific applications have used application-mediated checkpoint and restart techniques to deal with failures [6]. However, these schemes can be problematic in an environment where the interval between checkpoints is comparable to the MTBF.

In this paper, we describe possible approaches for the effective usage of multi-teraflop and petaflop systems. In particular, we present techniques that allow an MPI application to continue execution despite component failures. One of these techniques is diskless checkpointing, which enables more frequent checkpoints by redundantly saving checkpoint data in memory. Another technique uses inexpensive mechanisms to collect global status information, based on statistical sampling techniques. We also present a fault-injection methodology that enables systematic test and concrete evaluation of our proposed ideas. Finally, we describe plans to create an infrastructure for real-time collection of failure predictors.

The remainder of this paper is organized as follows. After presenting, in Section 2, experimental data to quantify the reliability problem, we describe our fault injection techniques in Section 3. We discuss fault-tolerance for MPI applications in Section 4 and system monitoring issues in Section 5. We show how to use such techniques for adaptive control, in Section 6, discuss related work, in Section 7, and conclude by describing our future work in Section 8.

2. Reliability of large systems

The mean time before failure (MTBF) of commodity hardware components continues to rise. PCs, built using commodity processors, DRAM chips, large capacity disks, power supplies and fans have an operational lifetime measured in years (i.e., substantially longer than their typical use). For example, the mean time before failure for today's commodity disks exceeds 1 million hours. However, when components are assembled, the aggregate system MTBF is given by $(\sum_{k=1}^{N} 1/R_k)^{-1}$, where R_k is the MTBF of component k. Intuitively, the least reliable component determines the overall system reliability.

This simple formula also assumes, optimistically, that component failures are independent. In reality, failure modes are strongly coupled. For example, the failure of a PC fan is likely to lead to other failures or system shutdown due to overheating.

Although component reliabilities continue to increase, so do the sizes of systems being built using these components. Even though the individual component reliabilities may be high, the large number of components can make system reliability low. As an example, Fig. 1 shows the expected system reliability for systems of varying sizes when constructed using components with three different reliabilities. In the figure, the individual components have one-hour failure probabilities of 10^4, 10^5, and 10^6 (i.e., MTBFs of 10,000, 100,000 and 1,000,000 h).

Even if one uses components whose one-hour probability of a failure is less than 10^6, the mean time for system failure is only a few hours if the system is large. By this assessment, the IBM/LLNL Blue Gene/L system, which is expected to have 65,368 nodes, would have an average "up time" of less than 24 h.[1] Moreover, these estimates are optimistic; they do not include coupled failure modes, network switches, or other extrinsic factors.

Because today's large-scale applications typically execute in batch scheduled, 4–8 h scheduled blocks, the probability of successfully completing a scheduled execution before a component fails can be quite low. Proposed petascale systems will require mechanisms for fault-tolerance, otherwise there is some danger they

[1] IBM has focused on careful engineering to reduce component and system failures below these levels in Blue Gene/L.

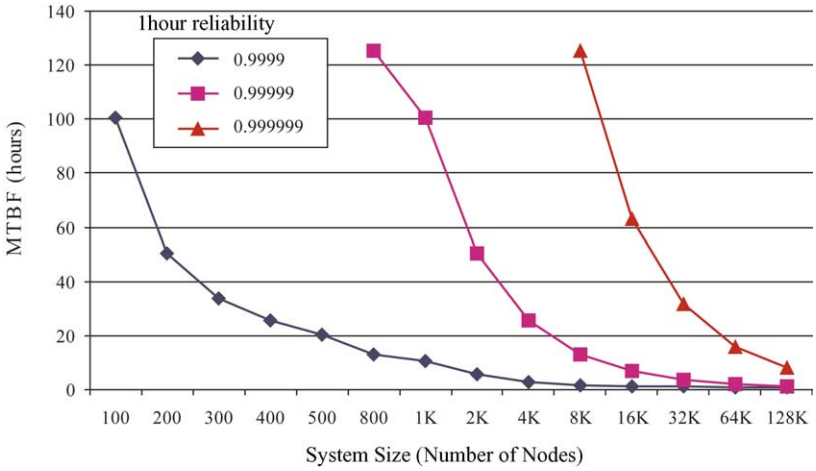

Fig. 1. System MTBF scaling for three component reliability levels.

may be unusable – their frequent failures would prevent any major application from executing to completion on a substantial fraction of the system resources.

As noted earlier, the data in Fig. 1 are drawn from a simplistic theoretical model. However, experimental data from existing systems confirm that hardware and software failures regularly disrupt operation of large systems. As an example, Fig. 2 depicts the percentage of accessible nodes on the NCSAs Origin Array, composed of 12 shared-memory SGI Origin 2000 systems. This data reflects outages that occurred during a 2-year period between April 2000 and March 2002.

For this system, the NCSA staff defined downtime as either scheduled (e.g. regular maintenance) or unscheduled (e.g. software or hardware halts). The overall availability is the ratio of total downtime to total time, whereas the scheduled availability is the ratio of total unscheduled downtime to scheduled uptime (i.e. total time minus scheduled downtime). For the NCSA system, hardware failures accounted for only 13% of the overall failures, whereas software failures represented nearly 59% of the total.

As this data shows, failures are already a noticeable cause of service disruption in today's systems. Given the larger number of components expected in petascale systems, such failures could make those systems unusable unless current approaches are modified. Hence, fault-tolerance mechanisms become a critical component for the success or large high-performance systems.

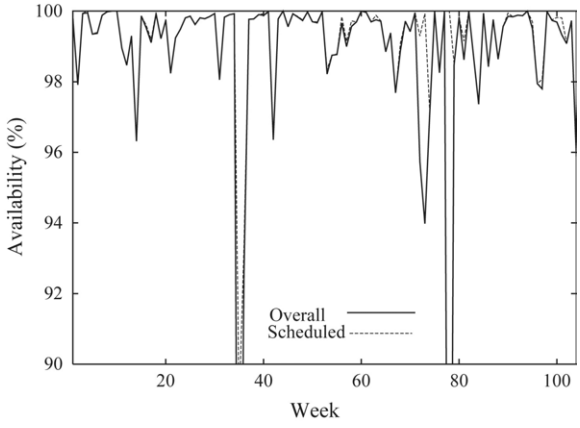

Fig. 2. NCSA Origin Array availability (system now retired).

3. Fault injection and assessment

Because the relative frequency of hardware and software failures for individual components is so low, it is only on larger systems that failure frequencies become large enough for statistically valid measurement. However, even in such cases, the errors neither systematic nor reproducible. This makes obtaining statistically valid data arduous and expensive. Instead, we need an infrastructure that can be used to generate synthetic faults and assess their effects.

Fault injection is the most widely used approach to simulate faults; it can be either hardware-based or

Table 1
Results from fault injection experiments (Cactus code)

Injection	Crash	Hang	Wrong output	Correct output	Total no. of injections
Text memory	49	12	6	933	1000
Data memory	12	31	0	959	1002
Heap memory	4	36	10	950	1000
Stack memory	82	43	0	877	1002
Regular registers	139	180	0	189	508
Floating point registers	0	10	10	480	500
MPI_Algather	15	0	5	70	90
MPI_Gather	17	16	41	0	74
MPI_Gatherv	0	0	23	13	46
MPI_Isend	0	0	0	90	90

software-based [11], and each has associated advantages and disadvantages. Hardware fault injection techniques range from subjecting chips to heavy ion radiation to simulate the effects of alpha particles to inserting a socket between the target chip and the circuit board to simulate stuck-at, open, or more complex logic faults. Although effective, the cost of these techniques is high relative to the components being tested.

In contrast, software-implemented fault injection (SWIFI) does not require expensive equipment and can target specific software components (e.g., operating systems, software libraries or applications). Therefore, we have developed a parameterizable SWIFI infrastructure that can inject two types of component faults into large-scale clusters: computation and communication. For each fault type, we have created models of fault behavior and produced instances of each. Thus, we can inject faults into parallel applications during execution and observe their effect.

Our fault injection infrastructure emulates computation errors inside a processing node via random bit flips in the address space of the application and in the register file. These bit errors can corrupt the data in memory or introduce transient errors in the processor core. In our experiments, we injected faults in three memory regions: the text segment, the stack and heap and the register file.

To maximize portability and usability across systems, we intentionally restricted fault injection to the application rather than the system software. Although memory and processor faults can be manifest in both, perturbing system address spaces is not feasible on the large-scale, production systems.

Communication errors, both hardware and software, can also occur at many levels, ranging from the link transport, through switches, NICs, communication libraries and application code. Many of the hardware errors can be simulated by appropriate perturbations of application or library code. Hence, our infrastructure injects faults using the standard MPI library. By intercepting the MPI calls via the MPI profiling interface, we can arbitrarily manipulate all MPI function calls. The infrastructure simulates four types of communication faults: redundant packets, packet loss, payload perturbation and software errors.

As an example of this fault injection methodology, we injected faults to the execution of a computational astrophysics code based on the Cactus package [9], which simulates the tri-dimensional scalar field produced by two orbiting stellar sources. We injected both computation and communication faults, and we classified their effects on observed program behavior. All tests were conducted on an x86 Linux cluster. Execution outcomes were classified as follows: an application crash, an application hang (i.e., an apparent infinite loop), a complete execution producing incorrect data, or complete execution producing the correct output.

Table 1 presents a portion of the observed results, corresponding to the effects of random bit flips in the computation and communication components of the Cactus code. Although the majority of the executions completed correctly, a non-negligible number of the injected faults resulted in some incorrect behavior. In particular, errors in the registers produced numerous crashes and hangs. Similarly bit errors in the MPI_Gather collective communication routine *always* caused some type of application error; this communication primitive carries data critical to program behavior. A more detailed experimental analysis can be found in [15].

4. Fault-tolerance in MPI codes

Historically, large-scale scientific applications have used application-mediated checkpoint and restart techniques to deal with failures. However, as we have seen, the MTBF for a large system can be comparable to that needed to restart the application. Consequently, the next failure can occur before the application has recovered from the last failure.

In general, application fault-tolerance techniques are of two types. In the first, the application is modified, and fault-tolerance is explicitly incorporated into the application's algorithms. Although flexible, this application-specific approach must be tailored to each application code. Alternatively, fault-tolerance can introduced in a semi-transparent way (e.g., via an underlying library).

For scientific applications, MPI is the most popular parallel programming model. However, the MPI standard does not specify either mechanisms or interfaces for fault-tolerance. Normally, all of an MPI application's tasks are killed when any of the underlying nodes fails or becomes inaccessible. Given the standard domain decompositions and data distributions used in message-based parallel programs, there are few alternatives to this approach without underlying support for recovery.

For the foreseeable future, we believe library-based techniques, which combine message fault-tolerance with application-mediated checkpointing, are likely to be the most useful for large-scale applications and systems. Hence, we are extending LA-MPI by incorporating diskless checkpointing [19] as a complement to its support for transient and catastrophic network failures. In this scheme, fault-tolerance is provided by three mechanisms: (a) LA-MPIs fault-tolerance communication, (b) normal, application-specific disk checkpoints and (c) library-managed, diskless intermediate checkpoints.

Disk checkpoints are under application control, and they write application-specific data for recovery and restart, based on application behavior and batch-scheduled computing allocations. Diskless checkpoints are intended to ensure that applications successfully reach their disk checkpoints (i.e., that node and other component failures do not prevent use of a typical 4–8 h batch-scheduled allocation with disk checkpoints every 0.5–1 h). Diskless checkpoints use the same interfaces as disk-based checkpoints, but occur more frequently. For example, if the disk-based checkpoint interval is 1 h, diskless checkpointing might occur every 15 min.

For diskless checkpointing, an application's nodes are partitioned into groups. Each group contains both the nodes executing the application code and some number of spares. Intuitively, enough extra nodes are allocated in each group to offset the probability of one or more node failures within the group. When an application checkpoint is reached, checkpoint data is written to the local memory in each node, and redundancy data (parity or Reed-Solomon codes [18] is computed and stored in the memories of the spare nodes assigned to each group.

If a node fails, the modified MPI layer directs the application to rollback and restart the failing process on a spare node, using data regenerated from checkpoint and redundancy bits. This scheme avoids both the long waits for job resubmission due to failures and the loss of data between disk-based checkpoint intervals. It also capitalizes on high-speed interconnects for rapid data movement, as a complement to lower-speed I/O systems.

To assess the efficacy of diskless checkpointing, we simulated this scheme for a hypothetical high-performance system containing 10,000 computation nodes with 500 spare nodes, under different configurations of spares per group. We assumed an inter-checkpoint interval of 30 min and a checkpoint duration between 0.5 and 2 min, proportional to the size of each group.

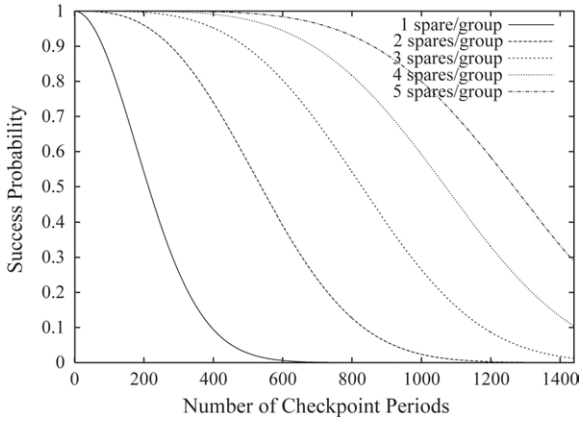

Fig. 3. Probability of successful execution with diskless checkpointing.

Fig. 3 shows the probability of successful execution (i.e., no catastrophic failure) as a function of application duration, expressed as checkpoint periods. Obviously, the longer the program runs, the more likely it will terminate due to component failures. Comparing the simulations with one and two spares per group, shows that assigning two spares per group can allow the application to successfully continue execution four times longer (at 90% successful probability). These preliminary results indicate that diskless checkpointing can significantly improve the application's tolerance to failures.

5. System monitoring for reliability

Many system failures are frequently caused by hardware component failures or software errors. Others are triggered by external stimuli, either operator errors or environmental conditions (e.g. [7]) predicts that the failure rate in a given system doubles with every 10 °C rise in temperature). Rather than waiting for failures to occur and then attempting to recover from those failures, it is far better to monitor the environment and respond to indicators of impending failure.

One attractive alternative is creation of an integrated set of fault indicator monitoring and measurement tools. As Fig. 4 shows, this fault monitoring and measurement toolkit would leverage extant, device-specific fault monitoring software to create a flexible, integrated and scalable suite of capture and characterization tools. As part of a larger research effort, we plan to integrate three sets of failure indicators: disk warnings based on SMART protocols [3], switch and network interface (NIC) status and errors, and node motherboard health, including temperature and fan status. Each set of failure indicator is described below.

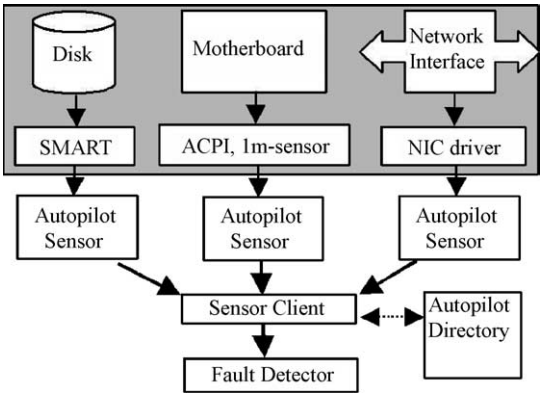

Fig. 4. Proposed fault indicator monitoring software.

5.1. Disk monitoring

Disk vendors have incorporated on-board logic that monitors disk status and health. This system, called Self-Monitoring Analysis and Reporting Technology (SMART), supports both ATA and SCSI disks. SMART typically monitors disk head flying height, the number of remapped sectors, soft retries, spin up time, temperature and transfer rates. Changes in any or all of these internal metrics can signal impending failure. Several open source implementations of SMART for Linux clusters provide access to disk monitoring and status data.

5.2. NIC and network status

Unusually high network packet losses can be a symptom of network congestion, network switch errors or interface card problems. Networks for multi-teraflop and petaflop systems contain many kilometers of cabling, thousands of network interface cards (NICs) and large numbers of network switches. Several common cluster interconnects support fault indicators. For example, the drivers for Myrinet [21] NICs maintain counters, accessible via user-level software, for a variety of network events, including packet transmissions, connection startups and shutdowns and CRC errors.

5.3. Motherboard status

For Linux systems, there are two popular approaches to assessing system health: the Advanced Configuration and Power Interface (ACPI) [1] and the lm_sensors toolkit [2]. ACPI is an open industry specification, co-developed by Hewlett-Packard, Intel, Microsoft, Phoenix and Toshiba, for power and device configuration management and measurement. In turn, the lm_sensors toolkit targets hardware and temperature sensors on PC motherboards. The toolkit takes its name from one of the first chips of this kind, the LM78 from National Semiconductor, which could monitor seven voltages, had three fan speed monitor-

ing inputs and contained an internal temperature sensor. Most other sensor chips have comparable functionality. ACPI and tools like lm_sensors serve complementary roles, providing data on both power consumption and motherboard health.

As Fig. 4 indicates, we envision capture of failure indicators from those three sources through sensors from our Autopilot toolkit [20]. Besides promoting portability across different systems, those sensors export data in a flexible format for analysis and fault detection.

Because collecting and analyzing data from every single node in a large system can become prohibitively expensive, we plan to adopt low-cost data collection mechanisms. One such mechanism where we obtained promising results uses statistical sampling techniques [16]. Instead of checking every system component individually, we select a statistically valid subset of components, inspect this subset in detail, and derive estimates for the whole system based on the properties found in the subset. The quality of these estimates depends on the components in the chosen subset. Statistical sampling provides a formal basis for quantifying the resulting accuracy of the estimation, and guides the selection of a subset that meets accuracy specifications.

In some cases, reducing the number of nodes for data collection may not be sufficient, as the amount of data produced in each node could still be excessively large. For these situations, we plan to implement data reduction via application signature modeling [14]. For each collected metric, we dynamically construct a signature that approximates the observed values for that metric. These signatures represent a lossy data compression of the original metrics that still captures the salient features of the underlying metric but are considerably less expensive than regular tracing techniques.

6. Intelligent adaptive control

Given the sources of faults described earlier, it is highly unlikely that a large system will operate without faults for an extended period. Thus, a long running application will encounter a system environment whose capabilities vary throughout the application's execution. To respond to changing execution conditions due to failures, applications and systems must be nimble and adaptive.

Performance contracts provide one mechanism for adaptation. Intuitively, a performance contract specifies that, given a set of resources with certain characteristics and for particular problem parameters, an application will achieve a specified performance during its execution [23]. To validate a contract, one must monitor both the allocated resources and the application behavior to verify that the contract specifications are met. Hence, the monitoring infrastructure must be capable of monitoring a large number of system components without unduly perturbing behavior or performance.

The notion of a performance contract is based on its analogue in civil law. Each party to a contract has certain obligations, which are described in the contract. Case law, together with the contract, also specify penalties and possible remediations if either or both parties fail to honor the contract terms. Witnesses and evidence provide mechanisms to assess contact validity. Finally, the law recognizes that small deviations from the contract terms are unlikely to trigger major penalties (i.e., the principle of proportional response).

Performance contracts are similar. They specify that an application will behave in a specified way (i.e., consume resources in a certain manner) given the availability of the requested resources. Hence, a performance contract can fail to be satisfied because either the application did not behave in the expected way or the resources requested were not available. Equally importantly, performance contracts embody the notion of flexible validation. For example, if a contract specifies that an application will deliver 3 gigaflops/processor for 2 h and measurement shows that the application actually achieved 2.97 gigaflops/processor for 118 min, one would normally consider such behavior as satisfying the contract. Intuitively, small perturbations about expected norms, either in metric values or in their expected duration, should be acceptable.

Combining instrumentation and metrics, a contract is said to be violated if any of the contract attributes do not hold during application execution (i.e., the application behaves in unexpected ways or the performance of one or more resources fails to match expectations). Any contract validation mechanism must manage both measurement uncertainty and temporal variability (i.e., determining if the contract is satisfied a sufficiently large fraction of the time to be acceptable). Reported contract violations can trigger several possible actions,

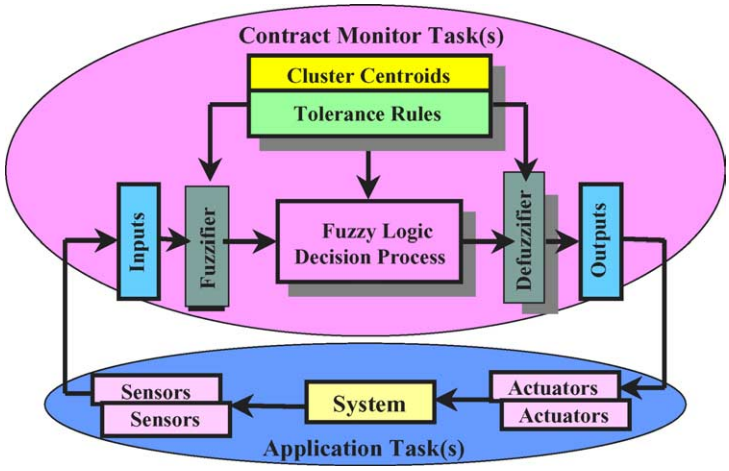

Fig. 5. Contract monitor infrastructure based on Autopilot.

including identification of the cause (either application or resource) and possible remediation (e.g., application termination, application or library reconfiguration, or rescheduling on other resources).

Autopilot is a toolkit for real-time application and resource monitoring and control built atop the Globus infrastructure. Using Autopilot, we have developed a contract monitoring infrastructure [12] that includes distributed sensors for performance data acquisition, actuators for implementing performance optimization decisions, and a decision making mechanism for evaluating sensor inputs and controlling actuator outputs. The main modules of this contract monitor are presented in Fig. 5.

In concert with development of the contract monitor, we created tools for visualizing the results of contract validation. As an example, Fig. 6(a) displays the contract output for an execution of the Cactus on a set of distributed resources. Each bar in the figure corresponds to the contract evaluation in one node. This node contract results from the combination of one contract for each measured metric. Using other controls in the GUI, the user can request visualization of values for specific metrics, as displayed in Fig. 6(b). Observed values correspond to points in the metric space, and rectangles represent the range of acceptable metric values determined in the contract.

In our first implementation of performance contracts, we captured metrics corresponding to computation and communication activity in the application. Computation performance was characterized by collecting values from hardware performance counters, through the PAPI interface [5]. For communication performance, we used the MPI profiling interface to capture message counts and data volumes. We are currently extending the set of possible metrics by including op-

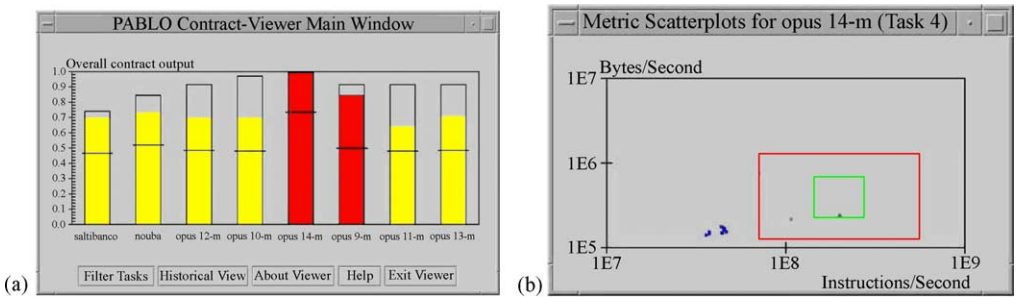

Fig. 6. Visualization of (a) node contract evaluation and (b) raw metric values.

erating system events, which can be captured by the Magnet tool [8].

7. Related work

Several research efforts are exploring more fault-toleran MPI implementations. These include Los Alamost MPI (LA-MPI) [10], which includes end-to-end network fault tolerance; compile-time and runtime approaches [4] that redirect MPI calls through a fault-tolerant library based on compile-time analysis of application source code; and message logging and replay techniques [13]. Each of these approaches potentially solves portions of the large-scale fault-tolerance problem, but not all. Compile time techniques require source code access, and message recording techniques often require storing very large volumes of message data.

Real-time performance monitoring and dynamic adaptive control have been used primarily with networks and distributed systems [22,17]. However, adaptation to failing resources must deal with poor performance, but also with conditions that can lead to application failure.

8. Conclusion

As node counts for multi-teraflop systems grow to tens of thousands, with proposed petaflop systems likely to contain hundreds of thousands of nodes, the assumption of fully reliable hardware and software becomes much less credible. Although the mean time before failure (MTBF) for the individual components is high, the large overall component count means the system itself may fail frequently. To be successful in this scenario, applications and systems must employ fault-tolerance capabilities that allow execution to proceed despite the presence of failures.

We have presented mechanisms that can be used to cope with failures in terascale and petascale systems. These mechanisms rest on four approaches: (a) real-time monitoring for failure detection and recovery, (b) fault injection analysis of software resilience, (c) support of diskless checkpointing for improved application tolerance to failures and (d) development of adaptive software subsystems based on the concept of performance contracts.

Looking forward, our preliminary tests with performance contracts can be extended both in spatial as in temporal scopes. On the spatial side, one could consider distributed processing of contract validation; different instances of a contract monitor would validate contracts for a subset of the nodes participating in a given execution. This distribution is essential to ensure scalability of the contract validation process.

From a temporal view, there are also several issues to be explored. The contract validation example that we presented was evaluated at a single moment. That validation does not consider previous application or system status. A more intelligent validation scheme should incorporate previous validations, and make decisions based on both current and past state.

Acknowledgements

This work was supported in part by Contract No. 74837-001-0349 from the Regents of University of California (Los Alamos National Laboratory) to William Marsh Rice University, by the Department of Energy under grants DEFC02-01ER41205 and DEFC02-01ER25488, by the National Science Foundation under grant EIA-99-75020, and by the NSF Alliance PACI Cooperative Agreement.

References

[1] Advanced Configuration and Power Interface Specification, V2.0a. http://www.acpi.info.
[2] Hardware Monitoring via lm Sensors. http://secure.netroedge.com/~lm78.
[3] B. Allen, Monitoring hard disks with SMART, Linux J. (2004).
[4] G. Bronevetsky, D. Marques, K. Pingali, P. Stodghill, Collective operations in an application-level fault tolerant MPI, in: Proceedings of the ICS'03, International Conference on Supercomputing, San Francisco, 2003, pp. 234–243.
[5] S. Browne, J. Dongarra, N. Garner, K. London, P. Mucci, A scalable cross-platform infrastructure for application performance tuning using hardware counters, in: Proceedings of the SuperComputing 2000 (SC'00), Dallas, TX, 2000.
[6] J. Daly, A model for predicting the optimum checkpoint interval for restart dumps, Lecture Notes in Computer Science 2660 (2003) 3–12.
[7] W. Feng, M. Warren, E. Weigle, The bladed beowulf: a cost-effective alternative to traditional beowulfs, in: Proceedings of the CLUSTER'2002, Chicago, 2002, pp. 245–254.

[8] M. Gardner, W. Feng, M. Broxton, A. Engelhart, G. Hurwitz, MAGNET: a tool for debugging, analysis and reflection in computing systems, in: Proceedings of the CCGrid'2003, Third IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003, pp. 310–317.
[9] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, J. Shalf, The cactus framework and toolkit: design and applications, in: Proceedings of the VECPAR'2002, Lecture Notes in Computer Science, vol. 2565, Springer, Berlin, 2003, pp. 197–227.
[10] R. Graham, et al., A network-failure-tolerant message-passing system for terascale clusters, in: Proceedings of the ICS'02, International Conference on Supercomputing, New York, 2002, pp. 77–83.
[11] M. Hsueh, T.K. Tsai, R.K. Iyer, Fault injection techniques and tools, IEEE Comput. 30 (1997) 75–82.
[12] K. Kennedy, et al., Toward a Framework for Preparing and Executing Adaptive Grid Programs, NGS Workshop, International Parallel and Distributed Processing Symposium, Fort Lauderdale, 2002.
[13] T. LeBlanc, J.M. Mellor-Crummey, Debugging parallel programs with instant replay, IEEE Trans. Comput. 36 (1987) 471–482.
[14] C. Lu, D.A. Reed, Compact application signatures for parallel and distributed scientific codes, in: Proceedings of the SuperComputing 2002 (SC'02), Baltimore, 2002.
[15] C. Lu, D.A. Reed, Assessing fault sensitivity in MPI applications, in: Proceedings of the SuperComputing 2004 (SC'04), Pittsburgh, 2004.
[16] C.L. Mendes, D.A. Reed, Monitoring large systems via statistical sampling, Int. J. High Perform. Comput. Appl. 18 (2004) 267–277.
[17] K. Nahrstedt, H. Chu, S. Narayan, QoS-aware resource management for distributed multimedia applications, J. High-Speed Networking 8 (1998) 227–255, IOS Press.
[18] J.S. Plank, A tutorial on Reed–Solomon coding for fault-tolerance in RAID-like systems, Software – Pract. Exp. 27 (1997) 995–1012.
[19] J.S. Plank, K. Li, M.A. Puening, Diskless checkpointing, IEEE Trans. Parallel Distribut. Syst. 9 (1998) 972–986.
[20] R.L. Ribler, J.S. Vetter, H. Simitci, D.A. Reed, Autopilot: adaptive control of distributed applications, in: Proceedings of the Seventh IEEE Symposium on High-Performance Distributed Computing, Chicago, 1998, pp. 172–179.
[21] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W. Su, Myrinet: a gigabit per second local area network, IEEE Micro 15 (1995) 29–36.
[22] J.S. Vetter, D.A. Reed, Real-time performance monitoring, adaptive control and interactive steering of computational grids, Int. J. Supercomput. Appl. High Perform. Comput. 14 (2000) 357–366.
[23] F. Vraalsen, R.A. Aydt, C.L. Mendes, D.A. Reed, Performance contracts: predicting and monitoring grid application behavior, in: Proceedings of the Grid'2001, Lecture Notes in Computer Science, vol. 2242, Springer, Berlin, 2002, pp. 154–165.

Dan Reed is the Chancellor's Eminent Professor at the University of North Carolina at Chapel Hill, as well as the Director of the Renaissance Computing Institute (RENCI), a venture supported by the three universities – the University of North Carolina at Chapel Hill, Duke University and North Carolina State University – that is exploring the interactions of computing technology with the sciences, arts and humanities. Reed also serves as Vice-Chancellor for Information Technology and Chief Information Officer for the University of North Carolina at Chapel Hill.

Dr. Reed is a member of President George W. Bush's Information Technology Advisory Committee, charged with providing advice on information technology issues and challenges to the President, and he chairs the subcommittee on computational science. He is a board member for the Computing Research Association, which represents the interests of the major academic departments and industrial research laboratories. He was previously Director of the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign, where he also led National Computational Science Alliance, a consortium of roughly 50 academic institutions and national laboratories that is developing next-generation software infrastructure of scientific computing. He was also one of the principal investigators and chief architect for the NSF TeraGrid. He received his Ph.D. in computer science in 1983 from Purdue University.

Charng-da Lu is a Ph.D. candidate at the Department of Computer Science, University of Illinois at Urbana-Champaign. He obtained his M.S. degree also at Illinois, in 2002. His main research interests are parallel computing, fault-tolerance and fault sensitivity analysis.

Celso L. Mendes is a Research Scientist at the Department of Computer Science, University of Illinois at Urbana-Champaign. Prior to his current position, he worked for several years as an Engineer at the Institute for Space Research in Brazil. He obtained the degrees of Electronics Engineer and Master in Electronics Engineering, both at the Aeronautics Technological Institute (ITA), in Brazil, and a Ph.D. in Computer Science at the University of Illinois. His main research interests are parallel computing, Grid environments and performance monitoring. Dr. Mendes is a member of the IEEE Computer Society and of the Association for Computing Machinery.