Invited Paper:

High availability for parallel computers

Dolores Rexachs and Emilio Luque
Computer Architecture an Operating System Department,
Universidad Autónoma de Barcelona, Barcelona 08193, Spain

ABSTRACT

Fault tolerance has become an important issue for parallel applications in the last few years. The parallel systems' users want them to be reliable considering two main dimensions, availability and data consistency. Availability can be provided with solutions such as RADIC, a fault tolerant architecture with different protection levels, offering high availability with transparency, decentralization, flexibility and scalability for message-passing systems. Transient faults may cause an application running in a computer system to be removed from execution, however the biggest risk of transient faults is to provoke undetected data corruption that changes the final result of the application without anyone knowing. To evaluate the effects of transient faults in the robustness of applications and validate new fault detection mechanism and strategies, we have developed a full-system simulation fault injection environment[1].

Keywords: Fault tolerance, availability, RADIC, transient faults, performability.

1. INTRODUCTION

To achieve more computing power it is usual to aggregate a large number of computing elements. The problem of this approach is that the more elements a system has, the probability of faults grows.

Recent trends in High Performance Computing (HPC) systems clearly indicate that future increases in performance, in addition to those resulting from improvements in multicore processor performance, will be achieved through corresponding increases in system scale. This growth in system scale, and the resulting component count, poses a challenge for HPC system and application software with respect to fault tolerance.

Fault tolerance has become an important issue for parallel applications running in parallel computer in the last few years. The miniaturization and the growth of the number of components, which form parallel machines, are the major root cause of the failures increasingly seen on these machines. The parallel machines' users want them to be reliable. Whereas availability refers to a system being in service, reliability refers to it performing correctly. Thus, there exists a fundamental distinction between reliable items and available items. When a reliable item fails, its life ends. When an available item fails, it can be repaired or otherwise returned to service after a relatively short down time. An available item oscillates all its life between the states "up" (working) and "down" (out of service) (Figure 1)

The improvement in computer reliability obtained by traditional methods is considered insufficient in many new installations, especially since computers are increasingly being used continuously and efficiently, a reliability increase can only be achieved by embedding redundant elements.

In order for the execution to complete correctly, parallel systems should use some fault tolerance strategy. In any case it is important to note that even with fault tolerance strategies, service interruptions (a complete stop of the program execution) may occur if data inconsistency or the system degradation generated by the faults reaches an unacceptable level. We will analyze some of the software methods that let parallel computers perform their intended function or at least keep their environment safe in spite of internal faults in hardware (persistent or hard, transient or soft errors including silent errors that can produce data inconsistency).

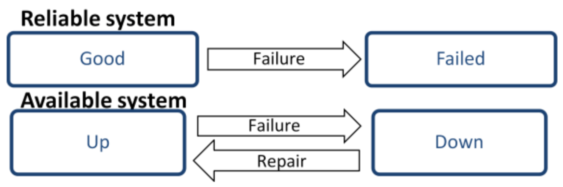
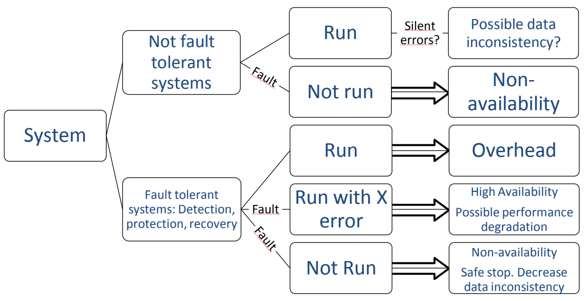


Figure 1. Reliable and available system

From a user's point of view, fault tolerance effects have two dimensions, availability and data consistency, as shown in Figure 2. Without fault tolerance, the execution is interrupted by one fault, but when fault tolerance is provided, the system can be maintained available with higher or lower degradation and with the possibility of detecting silent error reducing possible data inconsistency.

Fault tolerance represents a key issue in order to provide high availability in these parallel systems, because it


System
Not fault tolerant systems
Run
Silent errors?
Possible data inconsistency?
Fault
Not run
Non-availability
Fault tolerant systems: Detection, protection, recovery
Run
Overhead
Fault
Run with X error
High Availability
Possible performance degradation
Fault
Not Run
Non-availability
Safe stop. Decrease data inconsistency
Figure 2. Fault tolerant effects

[1] This work is supported by the MEC-Spain under contract TIN2007-64974

provides fault detection, protection and recovery. Fault tolerance can be provided in a parallel computer at three different levels [10]: hardware level, architecture level and application/system software level. The scope of this paper is in the application /system software level, where checkpointing techniques and rollback recovery are widely used to provide fault tolerance. As shown in Figure 3, there are different rollback-recovery protocols which can be useful to assure the application completion [6] [11]. The absence of a global clock in clusters makes it difficult to initiate checkpoints in all the streams of execution at the same time instance. We can use coordinated checkpointing or a message logging protocol.

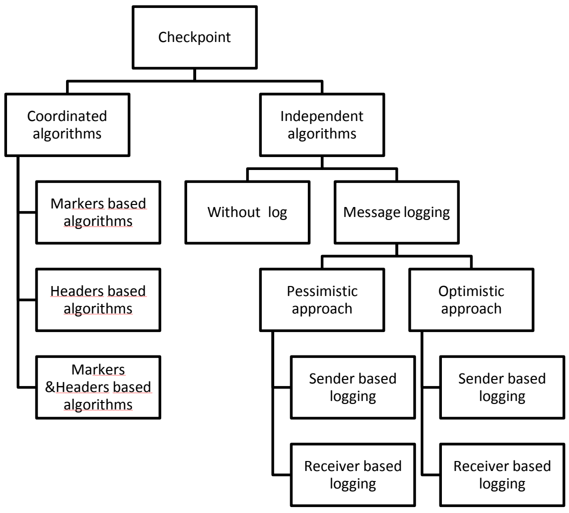


Figure 3. Rollback-recovery protocols tree in message-passing systems

A fault-tolerant system has the unique property that its overall reliability is higher than the reliability of its constituting parts. The secret of fault tolerance is how to structure these redundant components so that the failure of one does not bring the whole system down.
In order to achieve high availability, the challenges of a fault-tolerant system are to provide automatic and transparent fault detection, protection and recovery, which implies the evaluation of the appropriate quality indexes, as modeled in Figure 4. In addition we impose the constraint that the implementation of the proposed mechanisms will be done using only software solutions without requiring additional dedicated hardware.
Assuming a hypothesis that the effective performance of a high performance computer depends on its availability and that providing high availability implies a performance overhead, the study of the root causes of such overhead is necessary, including the performance degradation caused by faults.
Considering all these aspects, we proposed and developed RADIC (Redundant Array of Distributed Independent Fault Tolerance Controllers). RADIC is an architecture for providing Fault Tolerance (FT) in message-passing systems offering high availability with transparency, decentralization, flexibility and scalability for standard computer clusters with some node-local storage (hard disk, solid state disks –SSD- or partially dedicated main memory).

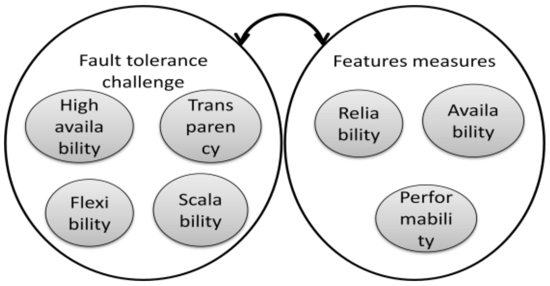


Figure 4. Fault Tolerance Challenges / Features measures

With the objective of analyzing transient faults effects' in computer systems' processor registers and memory, we have also developed an extension of COTSon [1], the HP's and AMD joint full system simulation environment. This extension allows the injection of faults that can change a single bit in processor registers and memory of the simulated computer. The developed fault injection system makes it possible to: evaluate the effects of single bit flip transient faults in an application, analyze the robustness of application against single bit flip transient faults and validate fault detection mechanism and strategies.
The remainder of this paper is organized as follows: The next section presents the basic concepts. Section 3 explains the protection levels currently implemented in RADIC architecture with some experimental results relating to them. More experimental results and the experimental environment are presented in section 4 and in section 5 we present a tool for analyzing soft errors, including silent errors that can produce data inconsistency. Finally, in section 6 we present our conclusions and future work.

2. FAULTS IN PARALLEL COMPUTERS: PROTECTION, DETECTION AND RECUPERATION

Computer clusters may be considered as a class of computing systems with degradable performance [13] i.e., under some circumstances during a determined utilization period, the system may present different performance levels. Such performance degradation is generally caused by faults occurrence, which may also affect the system availability if they have generated an interruption.
In order to achieve high availability, a fault-tolerant system must provide automatic and transparent fault detection, protection and recovery (Figure 5).
Until now, different efforts have been focused on providing high availability to computer clusters [3], [9]. The solutions resulting from these efforts are commonly based on rollback-recovery redundancy techniques [2], [4] and they have shown their efficacy in improving computer cluster performance and availability. In the

process of design and implementation it is necessary to take decisions that affect the trade-off between cost (resource use), performance and availability.

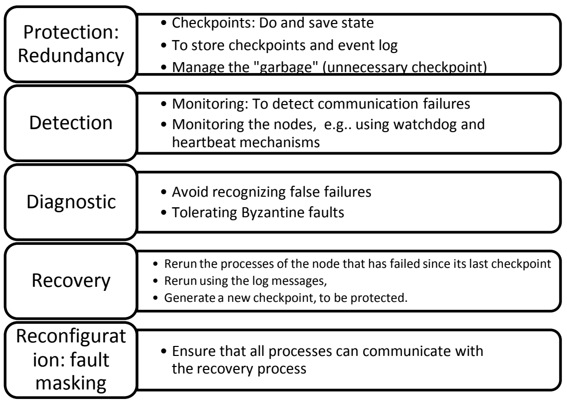

Protection: Redundancy
• Checkpoints: Do and save state
• To store checkpoints and event log
• Manage the "garbage" (unnecessary checkpoint)
Detection
• Monitoring: To detect communication failures
• Monitoring the nodes, e.g.. using watchdog and heartbeat mechanisms
Diagnostic
• Avoid recognizing false failures
• Tolerating Byzantine faults
Recovery
• Rerun the processes of the node that has failed since its last checkpoint
• Rerun using the log messages,
• Generate a new checkpoint, to be protected.
Reconfiguration: fault masking
• Ensure that all processes can communicate with the recovery process

Figure 5. Actions in fault tolerant systems

There is a correlation between performance and availability, such correlation is also known as performability. The fault tolerance mechanisms generate some kind of performance overhead because of their related activities, such as process state saving, messages exchange logging or system health monitoring. Performability, as the property of a computer system to deliver the performance required even though there are faults, is considered as a realistic, complete and accurate index for evaluating degradable systems such as computer clusters (Figure 6).

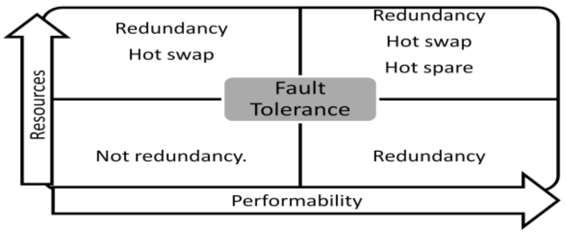


Figure 6. Parallel computer performability

Time To Failure (TTF) expresses the time to a fault or an error, even though it refers specifically to failures. Mean Time To Failure (MTTF) of a component expresses the amount of time elapsed between the last system startup or restart and then next error of the component. How these factors are involved in availability evaluation is shown in Figure 7. MTTF of a component is commonly expressed in years and it is obtained based on an averaged estimative of failure prediction done by the component's supplier.

Solutions to ensure a large MTTI (Mean Time To Interrupt) must also provide the means to restore the original system configuration (initial number of replacement nodes, or the process per node distribution) without stopping a running application. In addition to "reactive fault tolerance" (activities after a fault), it is also very desirable that it should perform preventive maintenance tasks by, for example, replacing fault-probable machines without system interruptions (Proactive Migration).

$$A = \lim_{t\to\infty} \frac{\sum up_{times}}{\sum(up_{times} + down\_times)} = \frac{MUT}{MUT + MDT} = \frac{MTTF}{MTTF + MTTR}$$

Availability (A)

Mean Time To Failure (MTTF)

Mean Up Time (MUT) (operational time)

Mean Down Time (MDT) (non-operational time)

Mean Time To Repair (MTTR)

Figure 7. Availability factors

A repairable item is defined by its availability. Stationary availability (A) is defined as the ratio between the sum of all operating times (MUT) and the useful lifetime (MUT+MDT).

When faults are taken into consideration, and these faults degrade the system's performance, performability measurements can be applied to evaluate a system in the presence of faults. Figure 8 depicts a chart exemplifying the throughput of an application, executed in an "on-line repairable" fault tolerant system, when single or concurrent faults occur against different degrees of availability (including no fault tolerance). This fault tolerance solution is characterized by keeping the system working but with the performance degraded. In this context, time constrained applications may not produce the expected results before their deadlines. In some cases, the degradation may reach unacceptable levels, leading to the need to perform a safe-stop and restart the entire system. Furthermore, the kind of fault uncovered by the availability degree may occur, interrupting the system, i.e., correlated faults when the availability degree only protects the system from single faults.

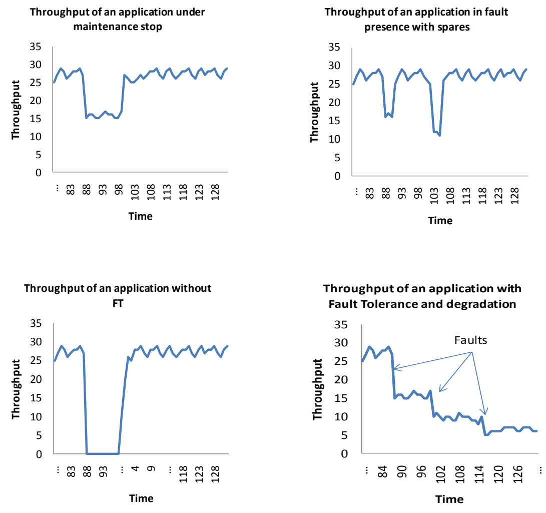


Figure 8: Throughput of an application in the presence of faults with different Fault Tolerance levels

Undetected errors, either hard or soft, are due to the lack of detectors for a component or the inability to detect it (e.g. transient effect too short). The real danger of undetected errors is that answers may be incorrect but the user wouldn't know it and they can also produce data inconsistency.

There are only a few publications showing evidence the occurrence of soft errors. The first evidence of soft errors were caused by contamination in the chips production in late 70's and 80's. Since 2000's, the reports of soft errors in large computer installations such as supercomputers and server farms are becoming more frequent. This happens because the number of components in this kind of installations is very large (thousands of CPU and terabytes of memory) and the powerful multi/many-core processors exhibit a high level of miniaturization (high density of transistors) and in consequence they are potentially less robust against transient faults. In Figure 9 (adapted from [12]) the possible outcomes of bit flip in a computer processor or memory are described.

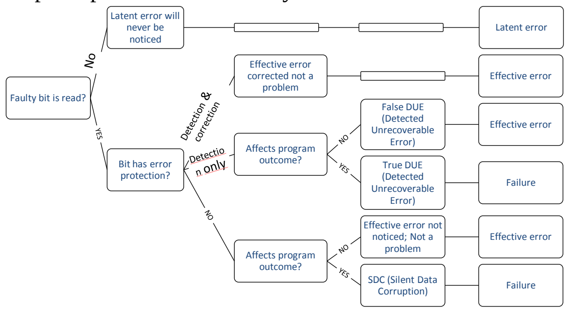

Faulty bit is read?
No
Latent error will never be noticed
Latent error
YES
Bit has error protection?
Detection & correction
Effective error corrected not a problem
Effective error
Detection only
Affects program outcome?
NO
False DUE (Detected Unrecoverable Error)
Effective error
YES
True DUE (Detected Unrecoverable Error)
Failure
NO
Affects program outcome?
NO
Effective error not noticed; Not a problem
Effective error
YES
SDC (Silent Data Corruption)
Failure

Figure 9: Classification of possible outcomes of a transient fault

For those systems requiring continuous operation over long times, having an "on-line repair mechanism" without disruption of the operation, with or without degradation of operation, is a key feature. Automatic reconfiguration can fail in case the of a second fault in the working units when it does not support simultaneous non correlated concurrent faults, although the presence of a maintenance system could reduces reliability, because of additional components. To deal with correlated concurrent faults requires more redundant elements.

3. THE RADIC ARCHITECTURE

RADIC (Redundant Array of Distributed Fault Tolerance Controllers) [4] is a fully fault tolerant architecture for message-passing parallel computers, providing high availability for parallel applications with transparency, decentralization, flexibility and scalability.

Our approach creates a "fully distributed controller" to manage faults in the nodes of the parallel computer. This controller contains two collections of dedicated processes, named protectors (P) and observers (O) (Figure 10), which collaborate to execute the fundamental tasks of a transparent fault tolerant scheme based on rollback-recovery: state saving, fault detection, recovery and fault masking. RADIC applies the uncoordinated checkpoint message-logging receiver-based technique for fault tolerance. The P and O processes collaborate as a fully distributed "parallel fault-tolerant manager" to automatically perform all activities required to ensure the correct ending of the parallel application in spite of failures in some nodes of the cluster.

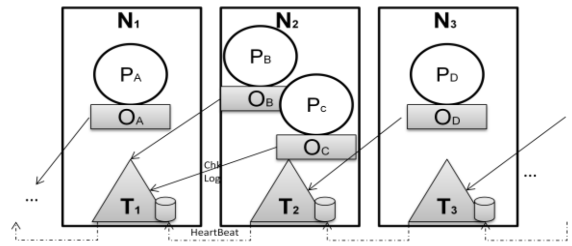


Figure 10. Example the RADIC architecture in a cluster. The arrows indicate the relationship between observers (O) and protectors (T).

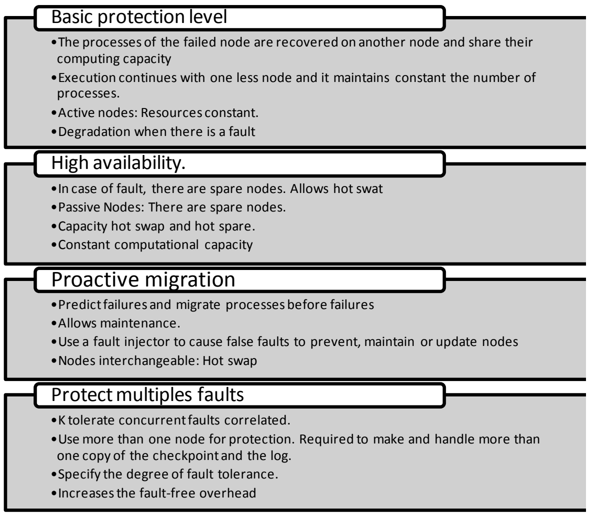
The set of observers (O) attached to the application processes, manage all delivered messages between application processes. Each observer also supports the checkpoint operation (chk) and the message-log mechanisms for the application process to which it is attached. The set of protectors (T) performs the failure detection task (Heartbeat–watchdog), operating like a distributed storage mechanism for the checkpoints and message-logs, and also recovering faulty processes.

The RADIC architecture acts as a layer between the fault-probable cluster structure and the message-passing library implementation. Such a layer performs fault-masking (message-delivering) and fault tolerance (checkpoint, event logs, fault detection and recovery) tasks.

RADIC is based on rollback-recovery techniques applying a pessimistic event-log approach. Such an approach was chosen because it does not need any coordinated or centralized action in order to provide its functionality, and as a consequence does not limit or reduce RADIC's scalability. RADIC considers any absence of expected communication as a fault but it can tolerates short transient faults by retrying the communication.

RADIC offers different protection levels, see Figure 11. In the Basic Protection level, RADIC operates without the need for any passive resource in order to provide its functionalities, using some active node of the configuration to recover a failed process which could lead to performance degradation. This protection level is well-suited for short-running applications, or applications that may tolerate resource loss, such as the dynamicly load balanced ones.

The High Availability level fits applications demanding a non-stop behavior. At this level, RADIC provides a flexible dynamic redundancy through a transparent management of spare nodes. This protection level avoids the system configuration change by incorporating transparent management of spares nodes. Moreover, it is possible to dynamically insert new replacement nodes during the program execution, allowing replacement of used spares or failed nodes. Such a feature increases the MTTI of the parallel application once the number of idle spare nodes remains constant, as it is shown in Figure 12.



Basic protection level

- The processes of the failed node are recovered on another node and share their computing capacity
- Execution continues with one less node and it maintains constant the number of processes.
- Active nodes: Resources constant.
- Degradation when there is a fault

High availability.

- In case of fault, there are spare nodes. Allows hot swat
- Passive Nodes: There are spare nodes.
- Capacity hot swap and hot spare.
- Constant computational capacity

Proactive migration

- Predict failures and migrate processes before failures
- Allows maintenance.
- Use a fault injector to cause false faults to prevent, maintain or update nodes
- Nodes interchangeable: Hot swap

Protect multiples faults

- K tolerate concurrent faults correlated.
- Use more than one node for protection. Required to make and handle more than one copy of the checkpoint and the log.
- Specify the degree of fault tolerance.
- Increases the fault-free overhead

Figure 11. RADIC protection levels

The Proactive migration fault tolerance is a useful characteristic in continuous running applications, where preventive maintenance stops are undesirable, so it became necessary to offer a mechanism allowing the performance of non-stop maintenance tasks [14].

The standard configuration of RADIC provides a protection degree which can tolerates several simultaneous non-correlated faults but if some component of the RADIC controller fails while that element is involved in the recovery of a fault, e.g. an observer and its respective protector, then the standard configuration is unable to support both faults.

To deal with this kind of correlated-concurrent faults, the RADIC architecture may be configured to increase the protection degree using more than one protector (P) for each application process, this is the Protect multiple faults level.

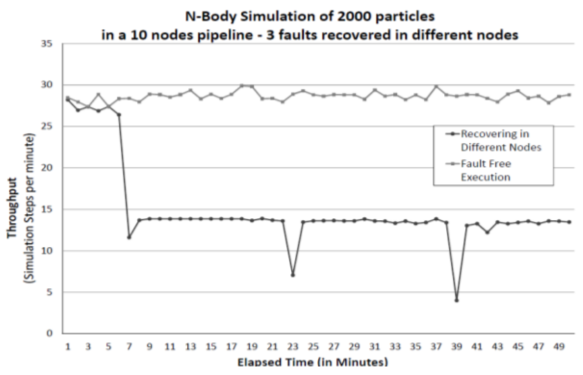
The flexibility of RADIC offers the possibility of modifying its configurations in order to accomplish the user’s requirements. RADIC permits the user to modify the checkpoint interval, observer/protector mapping, number of copies of each process, number and logical location of spare nodes, and the heartbeat/watchdog interval [8].

The RADIC architecture has been tested using a prototype called RADICMPI [5] and more recently a new version has been implemented over Open MPI [7] called RADIC/OMPI.

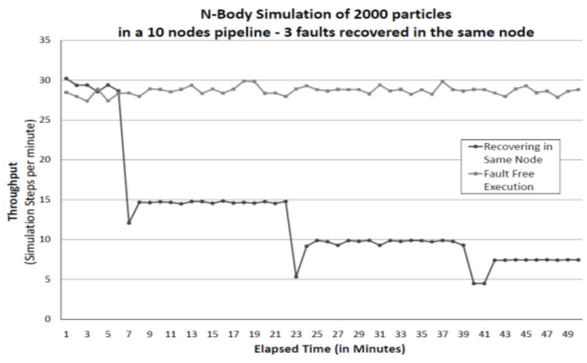
4. AVAILABILITY AND PERFORMANCE

Performability could be understood as the correlation between performance and availability when a rollback-recovery pessimistic message log-based fault tolerance protocol is applied into a computer cluster based on the message-passing model.

The root factors influencing the performability when using the RADIC fault tolerance architecture include: 1) redundant data replication (checkpoints and logs), 2) message delivery latency, because of the use of pessimistic logging and 3) process migration due to faulty nodes, because this can produce performance degradation.



(a)



(b)

Figure 12. Results of an N-Body simulation after three faults are recovered in: (a) different nodes and (b) the same node.

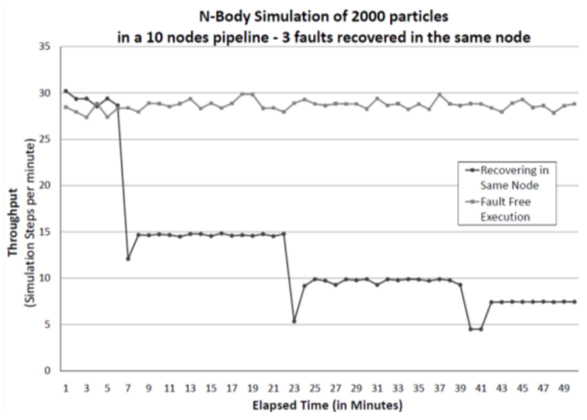


Figure 13. High Availability Protection Level: Results of an N-Body simulation after three faults are recovered in: spare nodes.

To analyze the impact of RADIC on system performance, it is necessary to measure the generated overhead in applications with different communication to computation ratio, because low or high ratios have different effects on the overhead. Execution Time also depends on the number of faults and the protection level.

In order to analyze the influence of RADIC on application performance, three class-D applications from the NAS benchmarks have been used: BT, LU and SP. These applications have been select due to their different communication to computation ratios. To evaluate the optimal checkpoint interval (I), we selected the equation $I = \sqrt{2k_0/\alpha}$ [10]. Where k_0 is the time spent creating and transferring checkpoint, and α is the probability of failure. Table 1 presents the calculated checkpoint intervals used to execute the above mentioned class-D NAS applications. In Figure 14 we can observe the overhead introduced by the message logging operation in some class C and D NAS applications, with and without RADIC and in the presence of faults. Depending on the application's communication to computation ratio the message logging can or cannot be completely overlapped with computation.

Table 1. Checkpoint intervals used to execute class-D NAS applications. Values are expressed in minutes and megabytes.

Applications: BT, LU, SP.	# Nodes	Running time (m)	Process size (MB)	Checkpoint interval (m)
BT D	16	43,79	1980	21,58
BT D	25	29,58	1400	16,28
SP D	16	55,01	1715	19,17
SP D	25	40,82	1251	14,90
LU D	8	103,84	1747	19,46
LU D	16	40,69	1061	13,13
LU D	32	20,63	722	9,91

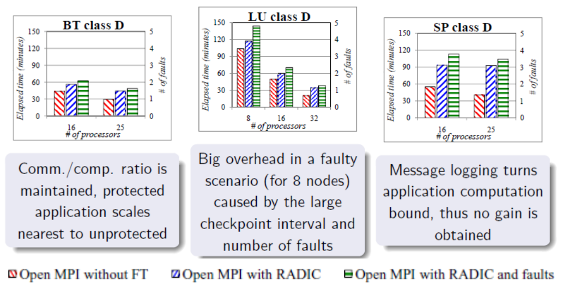


Figure 14. Execution time of class C and D NAS applications while using Open MPI with and without RADIC fault tolerance

5. SOFT ERROR AND DATA CONSISTENCY

Computer chip implementation technologies evolving to obtain more performance are increasing the probability of transient faults. The transient faults are those that may occur once and will not happen again the same way in the lifetime of that system. Transient faults in computer systems may occur in processor, memory, internal buses and devices, often resulting in an inversion in the state of a bit (single bit flip) at the fault location. Transient faults in computer systems are commonly the effect of cosmic radiation, high operating temperatures and variations in the power supply subsystem.

Transient faults may cause an application running in a computer system to be removed from execution (fail-stop) by the operating system when the change produced by the fault is detected by the processor or the operating system based on bad behavior of the application. In this case the transient fault would cause an application to misbehave (e.g. write into an invalid memory position; attempt to execute an inexistent instruction) which will then be abruptly interrupted by the operating system fail-stop mechanism. However, perhaps the biggest risk for applications is that transient faults provoke undetected data corruption and change the final result of the application, without anyone knowing. This data corruption happens when the transient fault bit-flip generates an incorrect final result that might not ever be noticed [12].

The risk of having transient faults affecting computation resulted in the need for researchers to have tools to simulate those faults to study their effects and also to test their theories and proposals. Since transient faults occur in a very unpredictable way, an environment with transient bit-level fault injection capabilities is needed to study the effects of these faults in the computer hardware stack (processor, buses, memory, etc) as well as the software stack (operating systems, middleware and applications)

As was mentioned in the introduction, we have developed an environment with single bit-level register level fault injection capabilities, based on COTSon [1].

The fault injection "campaign" description used in the environment only needs the value of three parameters to inject fault into a computer system simulation: a fault trigger, a fault location and a fault operation.

It is important to have an environment able to perform fault injection experiments in order to:

- Evaluate the effects of single bit flip transient faults on processor registers and memory on applications;
- Analyze application robustness against single bit flip transient faults on processor registers and memory;
- Test fault detection mechanisms: Be able to analyze the effectiveness of a fault injection through the application fault detection mechanism;

Our environment uses a full system simulator and allows both deterministic and non-deterministic fault injections campaigns and generates enough information about the fault injection to help in the further analysis necessary to build a better understanding of the effects of a transient fault in applications robustness and behavior.

By selecting a full system simulator as an environment to inject faults we also could achieve both precision and accuracy by having full control of the simulated computer processor.

With our fault injection environment we achieve a very transparent fault injection environment, as dealing with fault injection it isn't necessary to change the simulated computer operating system or the tested application.

With the developed environment, we were able to show evidence of the influence of compiler optimizations in the robustness of an application and of a fault detection mechanism against transient faults (Figure 15).

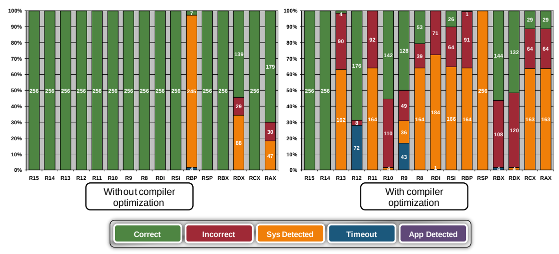


Figure 15. Some plot of the developed environment showing the impact generated in the application provoked injecting transient faults in different registers

6. CONCLUSIONS

The growth in HPC system scale poses a challenge in the design of automatic and transparent strategies and mechanisms to perform fault protection, detection and recovery. To evaluate the performance of these systems in the presence of faults, performability, the property of a computer system to deliver the performance required even though there are faults, is considered a realistic, complete and accurate index. RADIC, a fault tolerant architecture with different protection levels, can provide high availability with transparency, decentralization, flexibility and scalability for message-passing systems. The flexibility of RADIC offers the possibility of modifying its configurations in order to accomplish the user's requirements.

An increasing risk for applications is that transient faults, through silent errors, provoke undetected data corruption and change the final result of the application, without anyone knowing. We have developed a fault injection system for evaluating the effects of single transient faults in an application, analyzing the robustness of applications against these transient faults and validating new fault detection mechanisms and strategies.

Acknowledgments

This work has been possible thanks to the Fault Tolerance group of UAB, composed of Angelo Duarte, Guna Santos, Leonardo Fialho and Joao Gramacho.

7. REFERENCES

1. Argollo, E., Falcón, A., Faraboschi, P., Monchiero, M., & Ortega, D.: COTSon: infrastructure for full system simulation. SIGOPS Oper. Syst. Rev., Vol. 43 (Ed 1), pp. 52-61, 2009.
2. Bouteiller A., Herault T., Krawezik G., Lemarinier P., and Cappello F.: MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI. Int. J. High Perform. Comput. Appl. Vol. 20, no.3, pp. 319-333, 2006.
3. Chakravorty, S., Mendes, C. and Kale, L.V. Proactive fault tolerance in large systems. HPCRI Workshop in conjunction with HPCA 2005.pp 363-372, 2005.
4. Duarte, A., Rexachs, D., Luque, E.: Increasing the cluster availability using RADIC. Cluster Computing, 2006 IEEE International Conference on, pp. 1-8, 2006.
5. Duarte, A., Rexachs, D., Luque, E.: An Intelligent Management of Fault Tolerance in Cluster Using RADICMPI. LNCS Vol. 4192, Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 150-157, 2006.
6. Elnozahy E., Alvisi L., Wang Y., and Johnson D.: A Survey of Rollback-Recovery Protocols in Message Passing Systems. ACM Computing Surveys, vol. 34, no. 3, pp. 375-408, 2002.
7. Fialho L., Santos G., Duarte, A., Rexachs, D., Luque, E.: Challenges and Issues of the Integration of RADIC into Open MPI. LNCS Vol. 5759, Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 73-83, 2009.
8. Fialho L., Duarte, A., Rexachs, D., Luque, E.: Outcomes of the Fault Tolerance Configuration. CACIC 2009.
9. Engelmann C. and Geist A. Development of naturally fault tolerant algorithms for computing on 100,000 processors. http://www.csm.ornl.gov/~geist. 2002
10. Gropp, W., Lusk, E.: Fault Tolerance in Message Passing Interface Programs. Int. J. High Perform. Comput. Appl. 18(3), pp. 363–372, 2004.
11. Kalaiselvi S. and Rajaraman V.: A survey of checkpointing algorithms for parallel and distributed computers. Sadhana, vol. 25, no. 5, pp. 489-510, 2000.
12. Mukherjee, S. S., Emer, J., & Reinhardt, S. K.. The Soft Error Problem: An Architectural Perspective. HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, pp. 243-247, 2005.
13. Nagaraja, K., Gama, G., Bianchini, R., Martin, R. P., Meira Jr., W., and Nguyen. : Quantifying the Performability of Cluster-Based Services. IEEE Trans. Parallel Distrib. Syst. 16, 5, pp. 456-467, 2005.
14. Santos G., Duarte, A., Rexachs, D., Luque, E.: Providing Non-stop Service for Message-Passing Based Parallel Applications with RADIC. LNCS Vol. 5168, Euro-Par 2008, pp. 58-67, 2008.