SMCV: a Methodology for Detecting Transient Faults in Multicore Clusters

Diego Montezanti[1,3], Fernando Emmanuel Frati[1,3], Dolores Rexachs[2], Emilio Luque[2], Marcelo Naiouf[1] and Armando De Giusti[1,3]

[1]Instituto de Investigación en Informática LIDI, Facultad de Informática, UNLP
{dmontezanti,fefrati,mnaiouf,degiusti}@lidi.info.unlp.edu.ar
[2]Departamento de Arquitectura de Computadores y Sistemas Operativos, UAB
{dolores.rexachs,emilio.luque}@uab.es
[3]Consejo Nacional de Investigaciones Científicas y Técnicas

Abstract. The challenge of improving the performance of current processors is achieved by increasing the integration scale. This carries a growing vulnerability to transient faults, which increase their impact on multicore clusters running large scientific parallel applications. The requirement for enhancing the reliability of these systems, coupled with the high cost of rerunning the application from the beginning, create the motivation for having specific software strategies for the target systems. This paper introduces SMCV, which is a fully distributed technique that provides fault detection for message-passing parallel applications, by validating the contents of the messages to be sent, preventing the transmission of errors to other processes and leveraging the intrinsic hardware redundancy of the multicore. SMCV achieves a wide robustness against transient faults with a reduced overhead, and accomplishes a trade-off between moderate detection latency and low additional workload.

Keywords: transient fault, silent data corruption, multicore cluster, parallel scientific application, soft error detection, message content validation, reliability.

1 Introduction

The challenge of improving the computation performance of current processors has been achieved by increasing integration scale, which implies that the number of transistors within chips is growing. Additionally, the increment of the operation frequency has caused a raise in the internal operation temperature. These factors, added to a decrease in input power, cause processors to be more vulnerable to transient faults [14,17].

A transient fault is the consequence of interference from the environment that affects some hardware component in the computer. This can be caused by electromagnetic radiation, overheating, or input power variations, and can temporarily invert one or several bits of the affected hardware element (single bit-flip or multiple bit-flip) [2].

The way in which each transient fault occurs is unique; any given transient fault does not occur exactly the same never again throughout the lifespan of the system. These faults are short-lived and do not affect the regular operation of the system, although they can result in the incorrect execution of an application. Physically, they can be located anywhere in the hardware of the system; in this context, the faults that affect processor registers and logics are critical, since other parts of the system, such as memories, storage devices and buses, have built-in mechanisms (such as ECCs[1] or parity bits) capable of detecting and correcting this type of faults [1].

From the perspective of the program being run, the fault can alter the status of a hardware component that contains important information for the application. Depending on the time and specific location of the fault, it can affect application behavior or results and, therefore, system reliability [3].

The impact of transient faults becomes more significant in the context of HPC. Even if the mean time between faults (MTBF) in a commercial processor is of the order of one every two years, in the case of a supercomputer with hundreds or thousands of processors that cooperate to solve a task, the MTBF decreases as the number of processors increases. Since the year 2000, error reports due to large transient faults in large computers or server groups have become more frequent [1,20]. This situation is worse with the advent of multicore architectures, which incorporate a great degree of parallelism at hardware level. Also, the impact of the faults becomes more significant in the case of longer applications, given the high cost of relaunching execution from the beginning. These factors justify the need for a set of strategies to improve the reliability of high-performance computation systems. In this way, the first step is detecting the faults that affect application results but are not intercepted by the operating system and, therefore, do not cause the application to be aborted.

Traditionally, the existing proposals for providing transient fault tolerance have been divided into those that tackle the problem from a hardware standpoint, and those that do so from an application perspective.

Hardware-based techniques [8,9,11,13] aim to protect the various elements in the processor by adding additional logics to provide redundancy. These are most widely used in critical environments, such as flight systems or high-availability servers, where the consequences of a transient fault can be disastrous.

Hardware-redundancy-based techniques, however, are inefficient in general purpose computers. The cost of designing and verifying redundant hardware is high, and the environmental conditions in which the processors are used and processor ageing are the main causes for faults that cannot be predicted during the development stage. On the other hand, in many applications (audio or video on demand), the consequences of a fault are not as severe, so there is no critical need to add thorough fault-tolerance mechanisms [21].

The compromise between the achieved reliability and the resources involved makes software-redundancy-based strategies [19] to be the most appropriate for general purpose computational systems. The basic idea for detecting faults, called DMR[2],

---

[1] ECC: Error Correcting Code
[2] DMR: Dual Modular Redundancy

consists in duplicating application computation. Both replicas operate over the same input data and compare their outputs [8,11]. These techniques are characterized by their low cost and flexibility, allowing various configuration options to adapt to specific application needs [4].

An important aspect of detection lies in the validation interval. If results are compared only at the end, the fault that affects the application is detected with little additional workload, but the cost of relaunching the application from the beginning is high, especially in the case of large parallel applications. On the other end, if partial results are validated frequently, a high workload is introduced but the cost of re-executing the application from the last consistent state is lower than in the previous case. Therefore, a compromise must be reached between the detection interval and the additional workload introduced.

There are numerous proposals for detection, based on duplication, designed for serial programs, whose purpose is ensuring execution reliability. From this standpoint, a parallel application can be viewed as a set of sequential processes that have to be protected from the consequences of transient faults by means of the set of adopted techniques.

In this context, SMCV (Sent Message Content Validation) is presented, which is a proposal specifically designed for the detection of transient faults in scientific, message-passing parallel applications that execute on the nodes of a multicore cluster. SMCV uses software techniques that leverage the intrinsic redundancy existing in multicores, replicating each process of the parallel application in a core of the same processor. The detection is performed by validating the contents of the messages to be sent using a moderate validation interval and adding a reduced additional workload and a low overhead with respect to execution time. SMCV is a distributed strategy that improves the reliability of the system (formed by the cluster and the parallel application), isolating the error produced in the context of an application process and preventing it from propagating to the others. The end goal is to ensure that the applications that finish do so with correct results.

The rest of this paper is organized as follows: in Section 2, the theoretical context related to transient faults and their consequences in message-passing parallel applications is reviewed. In Section 3, related work is discussed. Section 4 describes this work´s proposal and explains the choices made. In Section 5, the methodology proposed is described in detail. Section 6 discusses the initial experimental validation. In Section 7, future lines of work are described, and Section 8 presents the conclusions.

2 Background

2.1 Soft Errors. Classification.

The errors (external manifestations of an inconsistent internal status) produced by transient faults are called soft errors. While transient faults affect system hardware, soft errors can be observed from the perspective of program execution.

Figure 1 shows the classification of the possible consequences of transient faults [24].

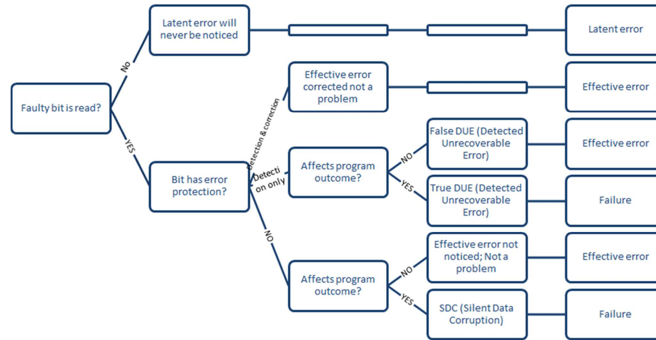

Fig. 1. Classification of possible outcomes of a transient fault (adapted from [24])

The soft error rate (SER) of a system is given by [18]:

$$SER = DUE + SDC + LF \quad (1)$$

A Detected Unrecoverable Error (DUE) is a detected error that has no possibility of recovery. DUEs are a consequence of faults that cause abnormal conditions that are detectable on some intermediate software layer level (e.g. Operating System, communication library). Normally, they cause the abrupt stop of the application. For instance, an attempt to access an illegal memory address (segmentation fault) or an attempt to run an instruction that is not allowed (e.g. zero division).

A Silent Data Corruption (SDC) is the alteration of data during the execution of a program that does not cause a condition that is detectable by system software. Its effects are silently propagated through the execution and cause the final results to be incorrect. From a hardware point of view, this is caused by the inversion of one or several bits of a processor´s register being used by the application, causing the program to generate faulty results.

A Latent Fault (LF) is a fault that corrupts data that are not read or used by the application so, despite the fault effectively happening, it does not propagate through the execution and has no impact on the results.

As a consequence, it is important that strategies are developed to intercept SDCs, which are the most dangerous type of faults that can occur from the point of view of reliability, because the program appears to be running correctly but, upon conclusion, its output will be corrupted.

2.2 Transient Faults in Message Passing Parallel Applications

The occurrence of a transient fault that causes an SDC in a core that is running one of the processes of a message-passing parallel application can have two different consequences:

$$SDC = TDC + FSC \quad (2)$$

A Transmitted Data Corruption (TDC) is an error in which the fault affects data that are part of the contents of a message that has to be passed. If undetected, the corruption is propagated to other processes of the parallel application.

On the other hand, in the case of a Final Status Corruption (FSC), the fault affects data that are not part of the contents of the message, but is propagated locally during the execution of the affected process, corrupting its final state. In this case, the behavior is similar to that of a sequential process.

Since a parallel application consists in the collaboration among multiple processes to perform a task, its success is based on communicating the local computation results obtained by each process to the others. Therefore, all faults that cause a TDC have a high impact on the end results. On the other hand, the faults that cause an FSC are related to the centralized part of the computation, and can therefore be detected by comparing the end results. Following this line, it follows that, if the task is divided among a larger number of processes, there will be a larger number of messages and a consequent growth in the TDC portion.

In this context, SMCV proposes a detection scheme that is focused on those faults that cause TDCs, and adds a final stage for comparing results to ensure system reliability. The solution proposed is discussed in Sections 4 and 5.

3 Related Work

Fault Tolerance (FT) involves three phases: detection, protection and recovery. One of the ideas most commonly used for detecting faults, proposed by Rotenberg [23], is duplicating the execution of a process hosted in a given core, using another core that works as redundancy. Both replicas operate on the same input data, compare their partial results every given period of time, and only one of them writes to memory or sends a message to another process [7,8,9,10,11].

Among the proposals that are based on software redundancy, code duplication, with several variants, has been the idea most widely adopted in the field of transient fault detection. SRT (*Simultaneous & Redundant Threading*) [5] is a first approximation to this, which consists in simultaneously running two replicas of a program as separate threads, dynamically scheduling hardware resources between them, and providing detection through input duplication and output comparison. In [6] CRT (*Chip-level Redundant Threading*) is proposed, which is the application of this technique to CMP environments. SRTR (SRT *with Recovery*) [7] proposes improvements to the detection mechanism and provides recovery through reexecution in the pipeline. CRTR (CRT *with Recovery)* [8] improves detection by separating execution from threads to mask the communication latency between cores, and it applies the recovery mechanisms proposed in [7] for a CMP environment. In [9], DDMR (*Dynamic* DMR) is proposed, a technique in which the cores that run the application in redundant mode are dynamically associated to prevent defective cores from affecting reliability, dealing with processing asymmetries and improving scalability. It introduces the possibility of configuring the system to operate in redundant mode or using

the cores separately for processing. All these solutions involve some modification to system hardware.

In [4], the *Mixed Mode Multicore* model is proposed, which allows running the applications that require reliability in redundant mode and, for applications that require high performance, avoiding this penalty, thus providing flexibility through configuration settings.

In [12], the proposal is obtaining a reduced version of the application by removing inefficient computation and computation related to predictable control flow. The full application and its reduced version are run in separate threads, providing redundancy and advance results that speed up the execution of the application. The authors in [11] propose selecting a core to carry out monitoring tasks over the processes that are run in the other cores, cyclically verifying their states. As an alternative, more than one core can be used for diagnosis operations, and the coverage level in case of faults can be configured, as well as the maximum overhead allowed. Thus, there is no need to produce a full replica of the program.

Among the solutions that are purely based on software, PLR [21] proposes the creation of a set of redundant processes for each application, being transparent to it. The implementation allows the Operating System to intelligently manage available hardware resources. This technique is designed for sequential programs.

In the context of these options, SMCV proposes a detection solution that is specific to message-passing parallel applications, not requiring any hardware modifications and leveraging the redundant resources that already exist in the multicore environment.

4 Work Hypothesis. Proposed Solution

In this section we present the rationale for SMCV. First, the usefulness of validating message contents is explained, and the features provided by the methodology are mentioned. Then, the leverage of redundant hardware resources by SMCV to increase system reliability is described.

4.1 Validating Contents of Sent Messages

The detection methodology proposed in this paper is essentially based on the hypothesis that, in a system formed by a multicore cluster that is running a message-passing parallel application, most of the significant computation (understood as that which impacts application results) will be part of the content of a message that is sent to other application process at some point during execution. Faults can corrupt data, flags, addresses or instruction code. However, if the corrupted value is significant for the results of the application, this situation will eventually be reflected on message incorrectness. Thus, of the total faults that can cause SDC, most will belong to the TDC category. Therefore, to detect faults that corrupt important data, the contents of the messages should be monitored. As regards the sequential phase, during which there are no communications, the end results are verified to ensure reliability.

SMCV is a detection strategy based on validating the contents of the messages to be sent. Each application process is duplicated, and both replicas compare all the fields that form message contents before sending; the message is sent only if the comparison is successful.

This technique allows detecting all faults that cause TDCs; from the point of view of the parallel application, SMCV ensures that any fault that affects the state of a process is not propagated to other process of the application, which confines the effects of the fault to the local process. Faced with an error, SMCV currently notifies the application and produces a safe stop. If a final comparison of the results is added to detect faults in the serial portion, SMCV ensures system reliability and, therefore, that the results of any application that finishes execution are correct.

Message contents are validated before sending the message. Thus, only one of the replicas effectively sends the message, which means that no additional network bandwidth is consumed. Taking into account that current networks have protocols that ensure reliable communications, there is no need to verify the contents of the messages upon reception (which would involve the transmission of two messages).

SMCV provides the following features:

- Each process and its replica are locally validated. The strategy is distributed in each application process. It is decentralized.
- It prevents the propagation of errors among application processes. Also, it detects errors in the serial part of the application by checking the end results.
- It introduces a low overhead in execution time, since only one comparison is added for each byte of each outgoing communication and the end result (it should be noted that the cost of comparison is lower than that of communication).
- A conservative detection strategy, designed for sequential programs, consists in duplicating application computation; to protect program outputs, each memory write operation is checked before being written [8]. Compared with this type of alternatives, SMCV involves a reduced work overload. In this sense, it can be said that it is a lightweight technique.
- When a fault is detected, the application is stopped, allowing relaunching the execution. There is no need to wait for the incorrect stop to re-execute, so SMCV narrows error latency. This carries a gain in reliability, but also in time, which becomes particularly significant in scientific applications that can run for several days.
- SMCV increases system reliability, understood as the number of times the application ends correctly, because it is able to detect faults that cause TDC.
- It achieves a trade-off between detection latency, additional workload and involved resources. SMCV allows latency in detection, since no verification is carried out when the corrupt value is first used. This postpones detection until the time when the altered data are part of the contents of the message. However, this implies a lower additional workload than validating each write operation (which produces low latency with high workload), and better leverages the resources than an only final comparison (which involves duplicating all computation to detect only at the

end, producing high latency with low workload). The less frequent communication between processes, the higher latency and the lower workload.

4.2 Leveraging Redundant Hardware Resources

Hardware manufacturer's trend is to add more cores to processors. However, many applications do not take advantage of all computation resources efficiently. On the other hand, the increase in the amount of transient faults goes hand in hand with the rise in the number of processing cores. As a consequence, the focus is no longer only processor performance, but factors such as reliability and availability have become more relevant. Therefore, the use of cores to carry out tasks related to fault tolerance has advantages both as regards to leveraging these resources as well as adding a beneficial feature for the system.

In this context, SMCV takes advantage of the intrinsic redundancy existing in multicores, using CMP cores to locate the replicas of the processes that perform useful computation for the application. The output to main memory is the critical aspect for selecting the cores that will be used to detect the faults that occur in the others. SMCV tries to exploit the memory hierarchy of the CMP, so that the redundancy of the computation that is executed in any given core is placed in another core with which some level of cache is shared. Thus, many comparisons will be resolved at LLC[3], minimizing main memory access.

5 Proposed Methodology Description

As already explained, SMCV is a software-centric strategy that can detect transient faults in a multicore cluster on which a message-passing scientific parallel application is being run. Upon detection of a fault, a user report is issued and the application is aborted, thus increasing system reliability.

Figure 2 shows an outline of the proposed detection methodology. Each process in the parallel application is run in a core of the CMP, and the computation it carries out is internally duplicated in a thread, which in turn is executed in a core that shares some cache level with the core running the original process. Thus, there is no need to access the main memory, taking benefit from the hierarchy to solve comparisons.

Each process is run concurrently with its replica, which means that a synchronization mechanism is required. When a communication is to be performed (point-to-point or collective), it temporarily stops execution and waits for its replica to reach the same point. Once there, all fields from the message to be sent are compared, byte to byte, to validate that the contents calculated by both replicas are the same. Only if this proves true, one of the replicas sends the message, ensuring that no corrupt data are propagated to other processes.

---

[3] LLC: Last Level Cache

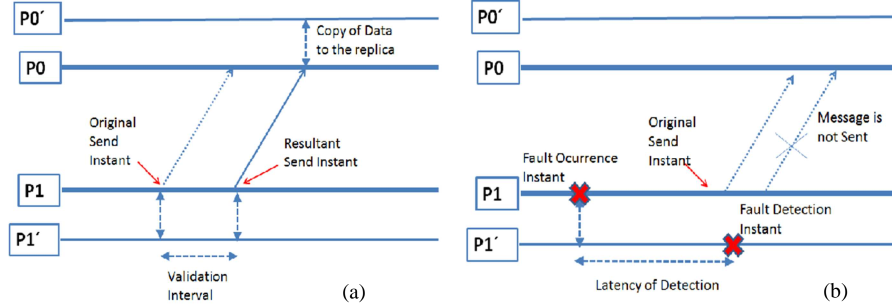

Fig. 2. SMCV methodology. (a) Proposed detection outline. (b) Behavior in presence of faults.

The recipient(s) of the messages stop upon reception and remain on hold. Once received, it copies the contents of the message to its replica (also using memory hierarchy) and both replicas continue with their computation. Assuming that network errors are detected and corrected at the network layer, the validated message reaches its destination uncorrupted. By comparing the message before sending it, the message can be sent only once. Were it be compared on reception, two copies of the message would have to be sent through the network, which would be detrimental to bandwidth use and network fault vulnerability.

Finally, when application execution finishes, the obtained results are checked once to detect faults that may have occurred after communications ended, during the serial part of the application.

5.1 Characterizing SMCV's Additional Workload

Additional workload is related to computing amount added by the fault detection strategy. This metric is useful to compare this methodology with other options. To have an approach, a conservative strategy based on the validation of memory write operations, similar to those used in sequential applications, has been analyzed. In this case, parallel application processes are also duplicated in threads as described, but the results of all write operations are validated (as opposed to validating only the contents of the messages sent). This strategy can detect all faults, but with a significant increase in computation amount.

The work overload W_{WV} introduced by the write validation technique is given by:

$$W_{WV} = (S + M.k)\,.\left(C_{sync} + C_{comp}\right) \quad (3)$$

In Equation (3), S represents the number of write operations performed by the application, excluding those corresponding to the messages it sends. It is assumed that the application sends M messages of k elements (average) each. C_{sync} and C_{comp} represent the costs of a synchronization operation and a comparison operation, respectively. The first factor in Equation (3) is therefore the total number of write operations performed by the application. If all write operations are validated, each will involve a synchronization operation and a comparison operation.

On the other hand, the workload added by message validation, W_{MV} is given by:

$$W_{MV} = M \,.\, (C_{sync} + k \,.\, C_{comp}) \quad (4)$$

In the case of message validation, for each message there is an only synchronization operation and k comparisons (one for each element in the message).

The relation between the workload introduced by SMCV and a strategy that validates all write operations will then be given by:

$$\frac{W_{MV}}{W_{WV}} = \frac{M \,.\, C_{sync} + M \,.\, k \,.\, C_{comp}}{S \,.\, (C_{sync} + C_{comp}) + M \,.\, k \,.\, C_{sync} + M \,.\, k \,.\, C_{comp}} \quad (5)$$

The quotient of Equation (5) is always a number lower than 1, which means that the additional computation overload for validating messages is lower than that for validating all write operations.

The analysis was carried out for one of the processes that communicate all its results. In the case of a process that performs serial computing, the overload for comparing the end results is added, but this is the same in both techniques. Therefore, this analysis is sufficiently general and representative of various situations.

It can be concluded that SMCV is a lightweight strategy that adds a reduced workload versus more conservative strategies that will detect faults that have no impact on the results of the application.

6 Initial Experimental Validation

The SMCV methodology has been assessed to determine its detection efficacy and the overhead introduced regarding to execution time. The results obtained are shown in this section.

6.1 Testing SMCV's Effectiveness

Tests were run with the detection tool to test its efficacy. The application used for the tests was a parallel matrix multiplication (C = A * B), programmed following the Master/Worker paradigm with 4 processes (the Master and 3 Workers), with the Master also taking part of the computation of the C matrix [22]. The Master process divides matrix A among all Workers and sends each one the chunk assigned to it, keeping a chunk for itself to participate in the calculation of the resulting matrix. Then, the Master sends each Worker a copy of the entire matrix B. After this, all processes compute their corresponding chunk of matrix C and, in the final stage, send the Master the part that they have calculated. The Master builds matrix C from what the Workers sent and its own computation. All messages used are non-blocking. The communications library used is OpenMPI.

All the experiments were run on a cluster with 16 blades, each one having 2 Quad Core Intel Xeon 5405 2GHz processors, 12 MB of L2 cache and 2 GB of main

memory. For this first test, an only blade was used, with the 4 processes and their replicas mapped to the 8 cores of the blade.

To implement SMCV, a part of an MPI communication primitive's library was developed, with the added functionality of fault detection by comparison upon sending, message contents duplication upon reception, and concurrency control between replicas. The Pthreads library was used for creating the replicas, and replica synchronization was done with semaphores.

The SMCV strategy was applied to the described application, replicating each of its processes in a thread as explained in Section 5 (for this, the source code of the application is required). The experiment consisted in injecting faults at various points of the application by means of a debugging tool. To do this, a breakpoint is inserted at a certain point of the execution of one of the application processes; the value of a variable is modified, and computation is resumed, so that the consequence of the fault at the end of the execution can be analyzed (this technique simulates a real fault in a processor register, since for data corruption to become apparent, it must be observable as a difference between the memory states of the replicas).

Even though a transient fault can randomly occur at any point during execution, significant processing time points were selected for the simulated injection, both for the Master and the Workers.

The strategy was capable of detecting all faults that affected message contents (TDC), as expected, notifying and aborting the application so that the corruption was not able to propagate. Thus, all Workers processing is protected. On the other hand, the faults that occurred in the data kept by the Master for local computation, and those that were produced after the partial results from all Workers had been collected by the Master in the last stage (corresponding to the FSC portion) were detected while comparing the end results.

6.2 Overhead Measurements

The overhead is a metric of the incidence of the detection tool on system performance, in the absence of faults. The overhead can be determined as the extra execution time implied by adding the SMCV strategy to the original application, on the architecture described above. The time added by SMCV is a consequence of the duplication of each process, the synchronization between replicas, the comparison carried out before each message is sent, the duplication of the messages received, and the final verification of the results.

Experiments were carried out by applying the SMCV methodology to the matrix multiplication application with 2, 4 and 8 processes (including the Master), with square matrix sizes of 512, 1024, 2048, 4096 and 8192 elements. The mapping between processes and processors was made in a way that ensures the same conditions of execution with and without the SMCV strategy, in order to directly compare the execution times. Up to 4 processes, an only blade was used, running application processes and its replicas using the all 8 cores. In the case of 8 processes, two blades were used, each of them running 4 processes of the application (without SMCV) and 4 processes and its replicas (8 cores) with SMCV.

Each experiment was run five times, and the results were averaged to improve stability. The standardized results, with respect to the execution time of the application with no fault detection, are shown in Table 1 and Figure 3.

	Procesos		
Tam (N)	2	4	8
512	0,87%	14,24%	55,11%
1024	0,01%	1,63%	21,40%
2048	0,39%	1,61%	10,05%
4096	-0,14%	0,91%	4,74%
8192	0,17%	0,92%	2,45%

Table 1. Overhead measurements

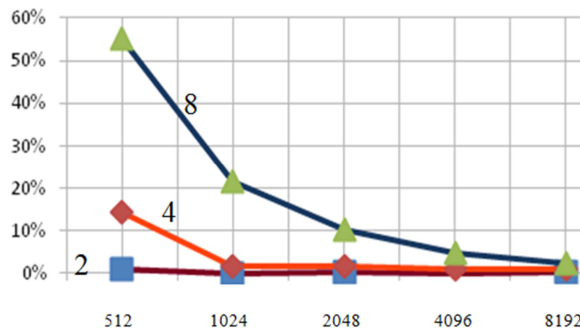

Fig. 3. SMCV's overhead in execution time

As it can be observed, the overhead decreases as the size of the problem grows up. This is because, with larger matrixes, the application spends more time computing. However, for any given number of processes, the number of messages remains constant. Therefore, synchronization, comparison and message contents duplication times are overshadowed by processing time. On the other hand, small matrixes require a short computation time and therefore all communication-related detection activities become more relevant.

Similarly, it can be seen that, for any given matrix size, overhead increases with the number of processes. This is explained by the fact that the number of messages (and therefore, synchronizations, verifications and copies) increases with the number of processes.

The case of 2 processes was the one that presented a wider dispersion between different repetitions of the experiment. A factor of randomness is present; inclusive, in the case of N = 4096, the incorporation of SMCV appears to perform better than the original application. However, with the precision of the obtained measurements, differences below 1 %, which occur in all cases, are considered negligible.

Based on the experiments carried out, it can be concluded that, when the size of the problem increases but the number of processes remains constant, the overhead is sig-

nificantly low. This would mean that, in real applications with high performance requirements, handling large amounts of data, similar overheads can be obtained.

7 Future Work

This work is part of a more extensive proposal whose purpose is providing transient fault tolerance for systems formed by scientific, message-passing parallel applications that are run on multicore cluster architectures.

Fault tolerance includes the phases of detection, protection, and recovery. In the context of permanent faults, the existing techniques most widely used are checkpointing and event log for protection, and rollback-recovery [24]. The proposal consists in integrating the transient fault detection methodology to the protection and recovery strategies available for permanent faults to provide transient fault tolerance. This means that there is no need of using triple modular redundancy (TMR) [16] with voting mechanisms to detect and recover from a transient fault. Also, since transient faults do not require system reconfiguration, recovery can be achieved by re-executing the same core of the failed process.

In the road towards achieving this goal, the following lines are open:

1. Perfecting the detection strategy:

- Expanding the experimental validation. A test that is more thorough than the one carried out so far requires the use of the methodology with standard applications. In the next stage, NAS benchmarks will be used, which are widely used in the scientific environment to measure the performance of parallel machines because they are representative of the type of computation most frequently made. These benchmarks respond to other parallel programming paradigms, and also have the advantage of providing self-verification functions of the results, which is useful for validating the detection strategy. In this sense, the integration with fault injection tools is desirable, to improve validation capabilities by means of extensive random fault injection campaigns. The overhead obtained with these applications will be measured.
- Achieving transparency for the application. At the current development level, SMCV's duplication process, based on threads, requires minor changes in the application code (and recompiling) to support the location of the replica in shared-memory with the original process and the use of the communications library with extended functionality. To obtain this transparency, replication must be implemented at the level of processes rather than threads.
- Optimizing the methodology to improve the trade-off between reliability, overhead, additional workload, detection latency (related to the recovery cost) and resource utilization. A detailed characterization will allow suggesting new ways of improving performance, considering the possibility of configuring the robustness level based on application coverage needs or maximum overhead permitted [6,11,13].

2. Providing full tolerance to SDC, restoring the system to its state previous to the fault:

In a following stage, the distributed detection strategy (already optimized) will be integrated with fault tolerance architectures oriented to permanent faults. The goal is obtaining a system capable of tolerating both permanent and transient faults. In this sense, integration with RADIC [15] will be attempted; RADIC is a transparent, scalable, flexible, and fully distributed architecture that provides fault tolerance through non-reliable elements and can recover after a permanent fault in a node. The aim is to leverage the methodology provided by RADIC for permanent faults (the rollback recovery mechanism, with non-coordinated checkpoints and message logs), and add transient fault tolerance. The resulting system will have to be tested to determine the reliability obtained, transparency for the application, resource utilization, overhead in absence of faults, and degradation in presence of faults.

8 Conclusions

In this paper, SMCV is presented, which is a transient fault detection methodology, purely implemented through software and specifically designed for scientific, message-passing parallel applications that are run on multicore clusters. Under the premise that in this type of applications, all information that is relevant for the end results is transmitted among the processes that are part of it, the SMCV strategy is based on validating the contents of the messages to be sent and comparing the end results to achieve a compromise between a high level of robustness against faults and the introduction of a low execution time overhead, consequence of the non-detection of the faults that would normally not affect the results. Also, it introduces a reduced additional workload versus the more conservative strategies that validate all write-to-memory operations, similar to the ones used in sequential applications.

References

1. 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, 243 – 247 (2005)
2. Wang, N. J., Quek, J., Rafacz, T. M., Patel, S. J.: Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks, 61 – 70 (2004)
3. Mukherjee, S. S.: Architecture Design for Soft Errors. Morgan Kaufmann (2008)
4. Lesiak, A., Gawkowski, P., Sosnowski, J.: Error Recovery Problems. Dependability of Computer Systems, 2007. DepCoS-RELCOMEX '07. 2nd International Conference on, 270 – 277 (2007)
5. Shivakumar, P., Kistler, M., Keckler, S. W., Burger, D., Alvisi, L.: Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks, 389 – 398 (2002)

6. Wells, P. M., Chacraborty K. Sohi G. S.: Mixed-Mode Multicore Reliability. ASPLOS 2009. SESSION: Reliable systems II, 169 – 180 (2009)
7. Reinhardt, S. K., Mukherjee S. S.: Transient Fault Detection via Simultaneous Multithreading. Proceedings of the 27th annual International Symposium on Computer Architecture, Vancouver, British Columbia, Canada, 25 – 36 (2000)
8. Kontz M., Reinhardt S. K., Mukherjee S. S.: Detailed Design and Evaluation of Redundant Multithreading Alternatives. Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02). Anchorage, Alaska, 99 – 110 (2002)
9. Vijaykumar T. N., Pomeranz, I. Cheng, K.: Transient-Fault Recovery using Simultaneous Multithreading. Proceedings of the 29th Annual International Symposium on Computer Architecture, Anchorage, Alaska. Session 3: Safety and Reliability, 87 – 98 (2002)
10. Gomaa M., Scarbrough C., Vijaykumar T. N., Pomeranz, I.: Transient-Fault Recovery for chip Multiprocessors. Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA ´03), San Diego, California, 98 – 109 (2003)
11. Golander A., Weiss S., Ronen R.: Synchronizing Redundant Cores in a Dynamic DMR Multicore Architecture. IEEE Transactions on Circuits and Systems II: Express Briefs Volume 56, Issue 6, 474 – 478 (2009)
12. Sundaramoorthy K., Purser Z., Rotenberg E.: Slipstream Processor: Improving both Performance and Fault-tolerance. ACM SIGPLAN Notices Volume 35, Issue 11, 257 – 268 (2000)
13. Barr A. H., Pomaranski K. G., Shidla D. J.: United States Patent Application Publication US 2005/0102565 A1: Fault Tolerant Multicore Microprocessing (2005)
14. Gramacho, J., Rexachs del Rosario, D., Luque, E.: A Methodology to Calculate a Program´s Robustness against Transient Faults. PDPTA 2011, 645 – 651 (2011)
15. Santos, G., Duarte, A., Rexachs del Rosario, D., Luque, E.: Providing Non-stop Service for Message-Passing Based Parallel Applications with RADIC. Euro-Par 2008, 58 – 67 (2008)
16. Mathur, F., Avizienis, A.: Reliability analysis and architecture of a hybrid-redundant digital system: generalized triple modular redundancy with self-repair. AFIPS '70 (Spring) Proceedings of the May 5-7, 1970, Spring Joint Computer Conference (1970)
17. Mukherjee, S.; Weaver, C.; Emer, J.; Reinhardt, S., Austin, T.: A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. MICRO-36.Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 29 – 40 (2003)
18. Weaver, C., Emer, J., Mukherjee, S. S., Reinhardt, S. K.: Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor, ACM SIGARCH Computer Architecture News, Volume 32, Issue 2, page 264 (2004)
19. Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., August, D. I.: SWIFT: Software Implemented Fault Tolerance, in Proceedings of the international symposium on Code generation and optimization, Washington DC, USA, 243–254 2005
20. Bronevetsky, G., Supinski, B.: Soft error vulnerability of iterative linear algebra methods. ICS ’08: Proceedings of the 22nd annual international conference on Supercomputing. New York, NY, USA: ACM, 155 – 164 (2008)
21. Shye, A., Blomstedt, J., Moseley, T., Reddi, V. J., Connors, D. A.: PLR: A software approach to transient fault tolerance for multicore architectures, Dependable and Secure Computing, IEEE Transactions on, Volume 6, Issue 2, 135 – 148 (2009)
22. Leibovich F., Gallo S., De Giusti L., Chichizola F., Naiouf M., De Giusti A.: Comparación de paradigmas de programación paralela en cluster de multicores: Pasaje de mensajes

e híbrido. Un caso de estudio. Proceedings of XVII Congreso Argentino de Ciencias de la Computación (CACIC 2011), 241 – 250 (2011)
23. Rotenberg E.: AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. Proceedings of the 29th Annual International Symposium on Fault- Tolerant Computing, 84 – 91 (1999)
24. Rexachs, D., Luque, E.: High Availiability for Parallel Computers. JCS&T Vol. 10 No. 3, 110 – 116 (2010).