Post-print of: Gramacho, Joao, et al. “Predicting robustness against transient faults of MPI based programs” in International journal of computational science and engineering (Ed. Inderscience), vol. 12, issue 2/3 (2016), p. 155-165. The final versión is available at DOI 10.1504/IJCSE.2016.076218

Predicting Robustness Against Transient Faults of MPI Based Programs

Joao Gramacho, Alvaro Wong, Dolores Rexachs, Emilio Luque
Computer Architecture and Operating Systems Department
Universitat Autònoma de Barcelona
Bellaterra (Barcelona), Spain
joao.gramacho@caos.uab.es, alvaro@caos.uab.es, dolores.rexachs@uab.es, emilio.luque@uab.es

***Abstract*— The evaluation of a program's behavior in the presence of transient faults is often a very time consuming work. In order to achieve significant data, thousands of executions are normally required and each execution will have the significant overhead of the fault injection environment. Our previously published methodology reduced significantly the time needed to evaluate the robustness of a program execution by exhaustively analyzing its basic blocks trace instead of using fault injection. In this paper we present an even forward improvement in the evaluation time of parallel programs robustness against transient faults by combining our methodology with PAS2P – a method that strives to describe an application based on its message-passing activity. The combination of our approach and PAS2P allowed us to predict the robustness of larger parallel programs, reducing in some cases in more than 20 times the time needed to calculate the robustness while obtaining a robustness prediction error of less than 4%.**

Transient faults, robustness, soft errors, reliability, PAS2P

I. INTRODUCTION

Computer chips are using smaller components, having more transistors, using those transistors with higher density and also operating at lower voltage. All these trends are increasing the computer processors die density and are responsible for the astonishing improvements in processing power of the last decades. Probably, as a side effect of such a scenario, these new powerful processors are becoming less robust than ever against transient faults [1].

Transient faults are those that might occur and may never happen again the same way in a system lifetime. Transient faults in computer systems may occur in processors, memory, internal buses and devices, often resulting in an inversion of a bit state (i.e. single bit flip) on the faulty location [2]. As common causes of transient faults in computer systems we can enumerate: cosmic radiation, high operating temperature and variations in the power supply subsystem.

There are three possible outcomes of a transient fault during a program execution: (i) no influence in the program execution at all; (ii) the application might misbehave (e.g. write into an invalid memory position; attempt to execute an invalid instruction); and (iii) an operating system's fail-stop mechanisms might be triggered, abruptly interrupting the faulty program. Nevertheless, an undetected data corruption is the biggest risk for a program execution. An undetected data corruption happens when the flipped bit – produced by the transient fault – generates an incorrect program result that might never be noticed. The errors that can indeed be noticed as effects of transient faults are called soft errors.

Usually, a fault injection environment is used to evaluate a program's behavior against transient faults. Those transient like fault injection environments are commonly based on architecture simulators or dynamic instructions instrumentation tools. In both cases they increase significantly (thousands of times in some studied cases) the time needed (wall clock) to run the program completely.

As we describe in section II.A, using fault injection to evaluate a program's robustness against transient faults requires statistical approximation and therefore thousands of executions (each execution of the same program with a distinct injected fault). The requirement of executing such vast amount of experiments limits the type of evaluated programs. The common case is to evaluate small benchmarks that need only fractions of seconds to run on ordinary computer architectures without simulation or instrumentation.

Section II.B shows the main idea behind our previously published methodology [3]. Our methodology is based on two steps: a basic block trace generation of the program during its execution and an analysis of the basic block trace backwards to evaluate the robust state of the processor registers on every single instruction executed by the program. Using this approach we were able to exhaustively evaluate a program's robustness against transient faults for a given processor architecture (taking into account the whole program execution) at an average 41% times faster than other methods based in fault injection and statistical approximation.

In this work we extend our previous approach by combining the methodology with PAS2P (Parallel Application Signature for Performance Prediction) [4]. PAS2P allows the prediction of the time needed to execute a parallel program based on Message Passing Interface (MPI) by running a small fraction of the program (its signature) and extrapolating the rest of the program execution based on a previously made analysis (briefly described in section III).

This combination allowed us to jump from evaluating programs that executed in order of hundredths of seconds to

This research has been supported by the MICINN Spain, under contract TIN2007-64974, the European ITEA2 project H4H, No 09011 and the Avanza Competitividad I+D+I program under contract TSI-020400-2010-120.

evaluating programs in the order of hundreds of seconds or more. Using this new approach we attained an error of less than 4% in the robustness prediction considering all evaluated programs. In our best case, this new approach took less than 4% of the time needed to perform a full program robustness evaluation.

In the section IV we explain the changes needed on both trace generation tools: the tool that traces the program basic blocks to compute robustness and the tool that traces the communication events to generate the PAS2P signature.

All the results obtained in our experimental evaluation are described in section V. We compare the time (wall clock) needed to calculate the programs' robustness and also the robustness obtained of a set of parallel programs that uses MPI both using the methodology presented in [3] (without PAS2P) and the method presented in this work.

Finally, in section VI we present our conclusion and considerations about this work and our next steps.

II. EVALUATING A PROGRAM ROBUSTNESS

Evaluating a program's behavior in presence of transient faults is often a very time-consuming work.

The usual method for transient faults evaluation is running the program in a fault injection environment thousands of times, injecting only one fault per execution and observing the program's behavior. All this work is done to achieve significant data in a statistical approximation. The fault injection environment adds significant overhead to each program execution by using architecture simulators or dynamic instrumentation of instructions to inject a fault.

A. *Executions in a fault injection environment*

At a fault injection environment, the evaluated program is executed and a fault in a form of a bit flip is injected on its architectural state (usually a bit in a processor register). At the end of the program execution, its result is evaluated to check the effect caused by the injected fault.

Whenever the program finishes correctly; i.e. its result is the same of a fault-free execution, the fault injected bit is classified as unACE (unnecessary for an Architecturally Correct Execution) [5]. If the program didn't finish correctly, presenting a result different from the fault-free execution, the fault injected bit is classified as ACE (necessary for an Architecturally Correct Execution) [5].

If the evaluated program has some kind of transient fault detection mechanism the changed bits may trigger the fault detection mechanism and lead the program to a fail stop condition, avoiding the propagation of the fault effect in the execution. In this case, as the execution finished accusing a fail stop condition, the program architectural bit changed is classified as DUE (Detected Unrecoverable Error) [5].

Changes in an ACE bit may lead to an abnormal program behavior and may also produce a different result than the obtained by a fault-free execution. It is common to classify those ACE bits as SDC (Silent Data Corruption) [5].

The reliability of a program against transient faults can be obtained by dividing the amount of unACE or DUE executions (free of failures) by the total amount of executions. The failure injection point is randomly chosen to ensure an even distribution.

The authors of [6] proposed a soft error detection mechanism based on source code transformation rules. In order to evaluate a given program with and without their fault detection mechanism, the authors performed 52,728 fault injection experiments in their fault injection environment.

In Error Detection by Duplicated Instructions (EDDI) [7], the authors reduced the amount of SDC cases by injecting instructions at compilation time. The injection aims to use free registers for redundancy, later adding verification for errors by comparing the original and redundant registers.

The authors evaluated their work by using eight benchmarks, each with 500 fault injection executions. Besides, four variations were used for each program (without protection plus three proposed mechanisms), achieving a total of 16,000 individual simulations to accomplish their work.

On Software-Controlled Fault Tolerance [8], the authors presented a set of software and hybrid (software and hardware) transient fault detection techniques. Each of the proposed techniques had a different cost/benefit relation by improving reliability or performance.

To evaluate the amount of SDC cases of an application with and without the proposed fault tolerance mechanisms, the authors executed fault injection experiments in a functional simulator. Faults were randomly injected and programs were executed until the end. In a total of ten sets of experiments, the authors evaluated the robustness of a set of benchmarks by simulating 5,000 executions with fault injection, except for two variations that simulated 1,000 executions. In each of the 504,000 simulated executions, a bit of a randomly chosen integer register of the IA64 architecture was flipped.

Continuing their research in fault tolerance for transient faults, the same authors of [8] proposed Spot [9], a technique to dynamically insert redundant instructions to detect errors generated by transient faults. This dynamic insertion was made in runtime using instrumentation.

Besides using a different architecture from the previous work (in [9] they used IA32 and protected only the eight general purpose 32 bit registers), the authors didn't use simulators. All the analysis and fault injections were made through an instrumentation tool. The authors evaluated 16 benchmarks and executed a total of 1.03 million fault injections to obtain their results (keeping 5,000 executions with fault injection per benchmark and configuration evaluated).

It is known that by using a fault injection based evaluation of robustness, the amount of executions will affect the robustness precision [8]. Also, simulators or dynamic instrumentation tools will increase time needed on each execution in comparison with a time spent by the program running directly in the architecture without instrumentation.

B. *Calculating a program's robustness against transient faults*

In our previously published work [3] we show that it is possible to calculate a program's robustness against transient faults in a two steps approach:

- In the first step we generate a basic block trace of the whole program execution shown in Fig. 1 (a). It, contains all programs basic blocks, the order these basic blocks are executed and information about the instructions of each basic block;
- In the second step, shown in Fig. 1 (b), we analyze this basic block trace and calculate the robustness of the program execution.

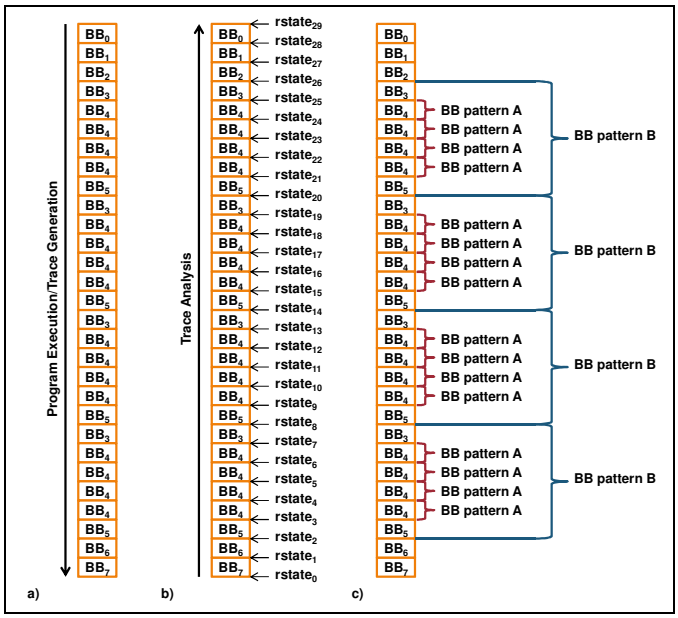

Figure 1. Basic block trace generation (a), trace analysis (b) and recognized basic block patterns (c).

In our trace generation tool (implemented as a PIN [10] tool using dynamic instructions instrumentation to log all basic blocks executed by the program) we've implemented a runtime compression algorithm.

This compression is based on finding basic block sequences repetitions, as shown in Fig. 1 (c), and changing the redundant data by just a small piece of information that will help us decompress the trace during the analysis, much like a run-length encoding.

1) *Robust state*

Robust state (defined in [3]) is a property of a processor register in a given point of a program execution and is represented by a vector of logical states (true or false) with as many states as the amount of bits of the processor register.

When an element of a register robust state vector is true, the register bit represented by the element is classified as unACE in the given execution point of the program.

A register bit flagged as true in its robust state means that any change in this register bit in the execution point of the program being analyzed won't be propagated to the rest of the program execution.

In the same way, when an element of a register robust state vector is false the register bit represented by the element is classified as ACE (we don't know yet if DUE or SDC) in the given execution point of the program.

A register bit flagged as false in its robust state means that any change in this register bit in the execution point of the program being analyzed can be propagated to the rest of the program execution.

$$1 \leq n < nins\left(Trace_{prog \times A}\right); i = f_{prog}(n) \tag{1}$$

$$f_{rstate}(reg, n) = [f_{rstate}(reg, n+1) \vee f_{wbits}(i, reg)] \wedge\sim f_{rbits}(i, reg)$$

The robust state f_{rstate} of a given register *reg* in a given program execution point *n* is a function of the robust state of the same register in the next instruction executed by the program, operated with the bits that the program instruction *i* (at point *n*) write on register *reg*, and then operated with the bits that the program instruction *i* (at point *n*) read from register *reg*.

The f_{wbits} and f_{rbits} are functions that will return, for a given instruction *i* and a given register *reg*, a vector of logical values with true in the vector elements that represents a register *reg* bit written (for f_{wbits}) or read (for f_{rbits}) by the instruction.

The function f_{prog} returns a processor architecture instruction *i* for a given trace execution point *n* that is lower than the amount of instructions *nins* in the trace generated by the program *prog* execution over an architecture *A*.

As every robust state calculation (except the last of the program trace) need the next program executed instruction robust state, the robustness analysis, as show in in Fig. 1 (b), is performed backwards, starting by the last executed program instruction (the last instruction of the last traced basic block) to the very first instruction executed by the program (the first instruction of the first traced basic block).

The robust state of the last instruction executed by the program will not be able to use its next instruction robust state as there is no next program instruction executed.

To solve this problem, we assumed that, at the end of the program execution, all processor registers bits can be assumed as robust (any change on them will not affect the program execution anymore).

$$n = nins\left(Trace_{prog \times A}\right); i = f_{prog}(n) \tag{2}$$

$$f_{rstate}(reg, n) = [f_{endstate}(reg) \vee f_{wbits}(i, reg)] \wedge\sim f_{rbits}(i, reg)$$

This particular case of the f_{rstate} function is only for the last instruction executed by the program being evaluated. We defined a function named $f_{endstate}$ that returns a vector with a robust state of a given register *reg* with all logical states as true (all register bits classified as unACE).

III. PAS2P

PAS2P [4] instruments a MPI program and executes parallel programs in a base machine, producing a trace log. The collected data is used to characterize the computation and communication behavior of the program. In order to obtain a machine-independent program model, the trace is logged using a logical global clock according to causality relations between communication events.

Once PAS2P generates the logical trace, it processes the trace using a technique that searches for similarity to identify and extract the most relevant phases and assign them a weight based on the number of times they occur. The signature will be defined by a set of phases and weights.

The execution of the signature in different target systems allows us to measure each phase execution time, and predict the program execution time in each target machine by extrapolating each phase's execution time using the obtained weights.

It is important to notice that the signature creation and execution is a two-step process:

- The first step is the analysis of the program, the building of the model and subsequent extraction of its phases and weights.
- The second step is the prediction method where PAS2P executes the signature in a target machine to measure the phases' execution time and predict the program execution time.

A. *Parallel application model*

To create the signature, first PAS2P build a model (Machine-Independent Model) of the application and then use that model to perform the predictions.

By instrumenting the MPI program, PAS2P obtain a program communication and computation trace that contains all the communications events between processes and computation time elapsed between MPI primitives.

In this context, an event will be a message sent or a message received. With this information we build the program model and use it to study at what point of the program the most computing time is spent (relevant phases), and how many times those phases are repeated (weights).

In the Fig. 2 (a) we show an example of a hypothetic program trace generated by PAS2P, with the phases recognized as P_0, P_1, P_2, P_3, P_4, P_5, P_6 and P_7.

The result of the PAS2P trace analysis is presented on Fig. 2 (b), showing all program phases ordered by its relevancy in the total program execution time. In this example PAS2P has detected that only two of the program phases (P_1 and P_2) are the most relevant to the execution time prediction.

In the last step of its analysis PAS2P determines a way of executing the minimum fraction of the program, as shown in Fig. 2 (c), which allows measuring the time needed to execute an amount of those phases assumed most relevant. PAS2P will try to determine the measurement start and finish by leaving some warming up phases before the measurement. PAS2P will also try to run a maximum of 100 phase's repetitions to be able to calculate a good average time for one phase. These measured averages will be used with the weights of each phase to predict the total program execution time.

B. *Performance prediction*

The executable signature runs the parallel program from the beginning and measures the time spent from the point a phase begins until its end. When a phase has been measured, PAS2P continues the program execution until a new relevant phase is found.

The signature repeats this method and proceeds to execute all constituent phases. When the last phase has been measured, the signature finalizes its execution that is often just a small fraction of the whole program execution.

The prediction of the program total execution time is a matter of adding the multiplication of each phase execution time by its weight as:

$$\text{PET} = \sum_{i=1}^{k} PhaseT_i \times W_i \qquad (3)$$

Where PET is the predicted execution time, k is the number of phases, $PhaseT_i$ is the phase i execution time and W_i is the phase i weight.

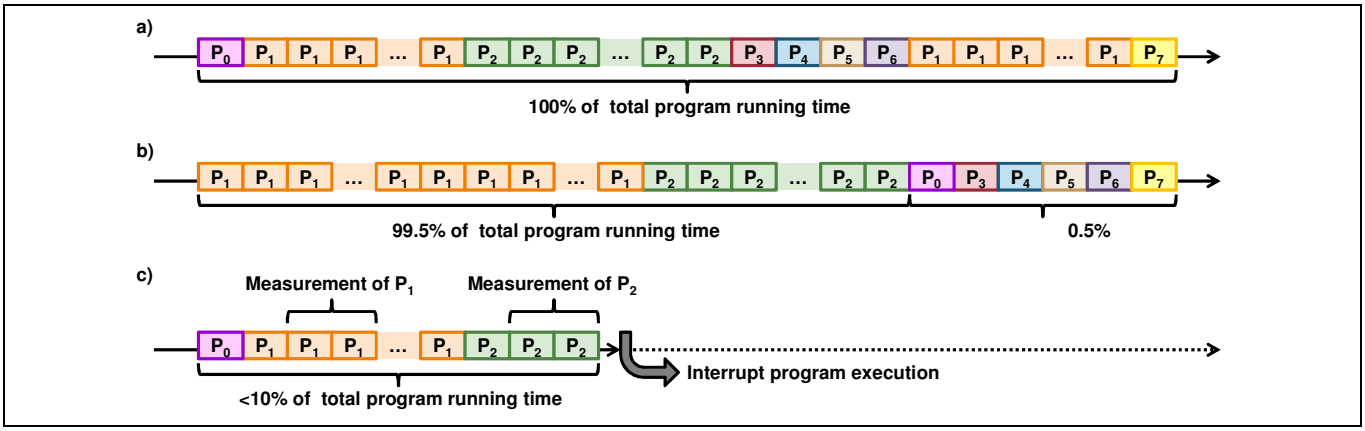

Figure 2. PAS2P trace generation (a), analysis (b) and signature execution (c).

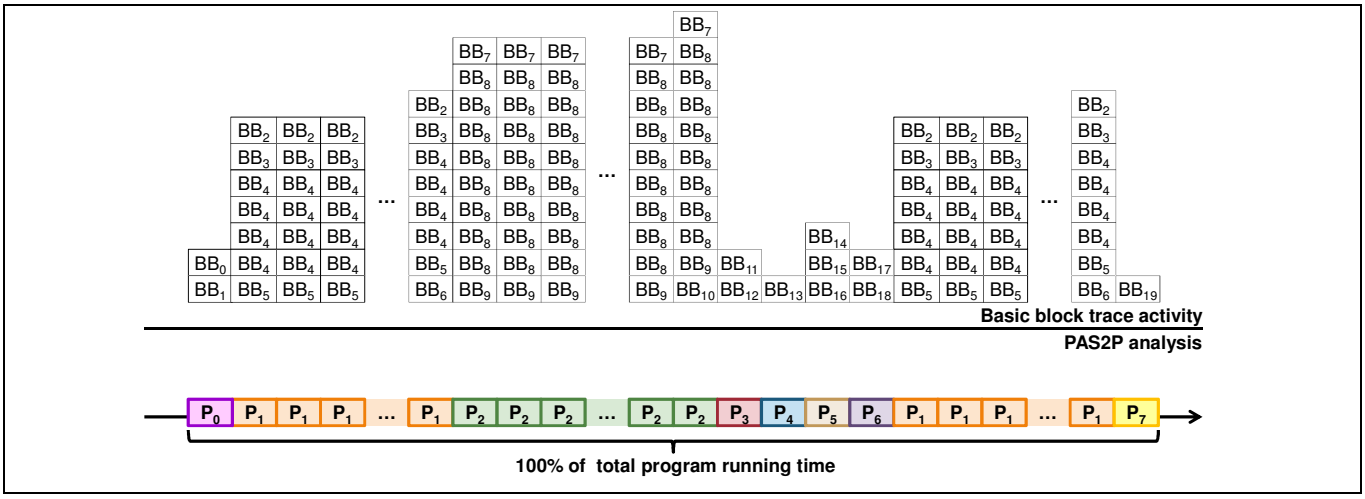

Figure 3. Basic block trace and PAS2P analysis phase's example.

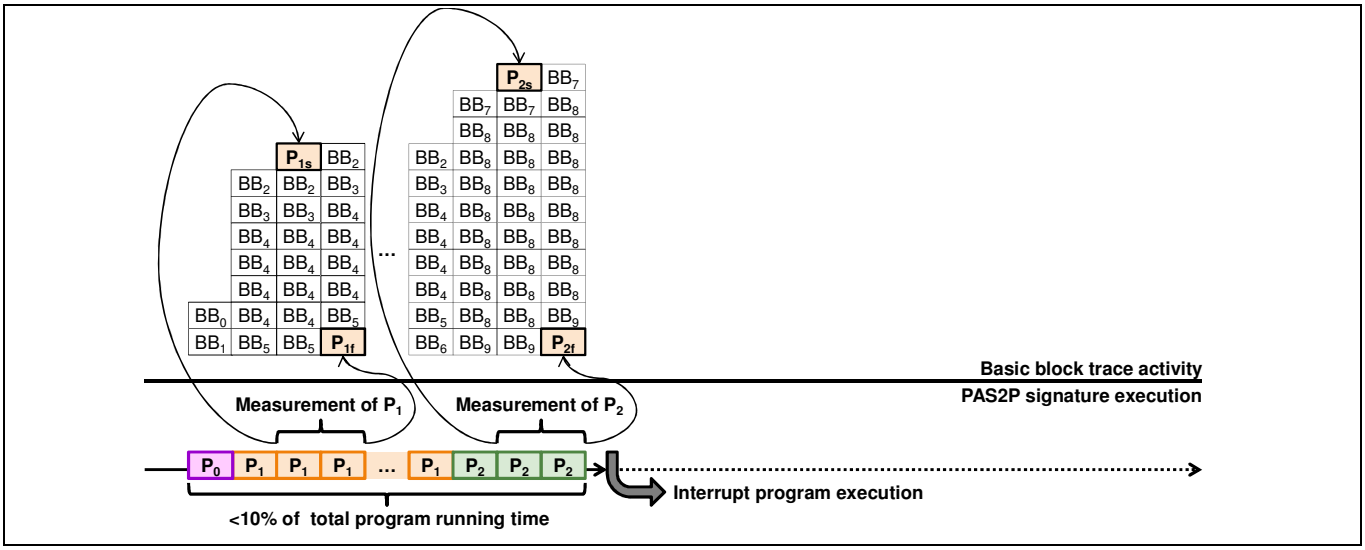

Figure 4. Basic block trace of a PAS2P signature execution.

IV. EVALUATING A PAS2P SIGNATURE ROBUSTNESS

In order to combine both the robustness evaluation methodology and PAS2P methodology we changed the trace generation tools (the one that generates the basic block information for the robustness analysis and the one that generates PAS2P phase's information) to cooperate during their execution.

In Fig. 3 we present an example of a basic block trace activity associated to the phases analyzed by PAS2P during the hypothetical program execution.

In this example, P_0 to P_7 are PAS2P recognized phases of the program and BB_0 to BB_{19} are basic blocks executed by the program.

The basic blocks trace activity shown in Fig. 3 will have all basic blocks executed by the complete execution of the evaluated program.

The analysis of this trace can calculate either the whole program execution robustness or a per basic block robustness. In both cases the robustness evaluated will be informed after analyzing the whole basic block trace file.

Our strategy to combine both methodologies (and tools) was to make PAS2P inform the basic block tracing tool about the beginning and the end of each measured phase.

With this interaction between the tools, the trace generated with basic block information stores two new types of information:

- A phase start tag in the beginning of every phase being measured by PAS2P;
- A phase finish tag in the end of every phase being measured by PAS2P.

In Fig. 4 we present a trace activity of a PAS2P signature execution. During this evaluation, PAS2P will inform the basic

block tracing tool by shared memory the phase that is starting or finishing its measurement. The basic block tracing tool, then, put this information as tags in the trace.

The tool that performs the robustness evaluation based on a basic block trace had to be changed too. The new version computes the program robustness the same way as before, but also presents a summary information of the specific robustness of each program phase tagged by PAS2P.

With this summary, and multiplying it by each phase weight informed by the PAS2P analysis, we could predict the whole program robustness with just a fraction of its real execution (the execution of the PAS2P signature). The robustness analysis is performed individually for each process of the parallel program.

The current version of our methodology treats each parallel program process as an individual program that starts, runs and finishes its execution. One basic block trace is generated per each programs processes executed.

V. EXPERIMENTAL EVALUATION

In order to realize our experimental evaluation we designed a set of experiments to calculate the robustness against transient faults of five programs. Our methodology is applied to both the standard program execution and the PAS2P signature execution of the program being evaluated.

The selected programs are part of the NAS Parallel Benchmark in its version 3.3. Because PAS2P required an MPI based parallel application, we choose to evaluate the MPI versions of **BT**, **CG**, **FT**, **LU** and **SP** benchmarks with their B class.

In comparison to our previously published work, where we evaluated the S class of the same five benchmarks, we are scaling our robustness analysis from programs that executed in tenths of seconds (0.16 on average) using a single processor to programs that execute in hundreds of seconds (167.54 on average) of wall clock time using four parallel computing nodes with the same processors of the previous work.

All five benchmark programs used in this experimental work were compiled using GNU C and Fortran in their version 4.4.1, with static linkage of libraries used by the programs and with maximum code optimization during compilation (O3). Also, all five benchmarks were compiled to run dividing they work between four computing nodes.

The four computing nodes used in the experiments have Linux Ubuntu Server operating system in version 9.10 with 64 bits kernel in version 2.6.31. The version of the OpenMPI library used was 1.4.3. The hardware of all computing nodes used have one 2 GHz AMD Athlon 64 X2 processor with 2 gigabytes of memory.

A. *PAS2P signature generation*

In the first step of our experimental design we've generated all PAS2P signatures of the programs being evaluated. We also tested the PAS2P prediction tool just to evaluate the prediction of the execution time of the programs.

Fig. 5 shows the time (wall clock) needed (on average) to completely execute the programs (without any instrumentation) and the time needed to execute the PAS2P signature.

Fig. 6 shows the error in the predicted execution time of the evaluated programs by comparing it with the standard program execution time.

Fig. 7 shows the trace generation overhead (also in comparison with the standard program executions) of the evaluated programs.

The average error of the predicted execution time was 2.88% (1.81% without taking into account the **FT** benchmark) and the average overhead in the execution time was 9.84% (4.12% without taking into account the **FT** benchmark).

In both figures (6 and 7), the FT benchmark presented a particular behavior, scoring worse than the other programs evaluated. The problem with the **FT** benchmark is that the workload used to evaluate the program was small enough to allow the PAS2P prediction tool to find phases with enough repetitions to have an accurate evaluation.

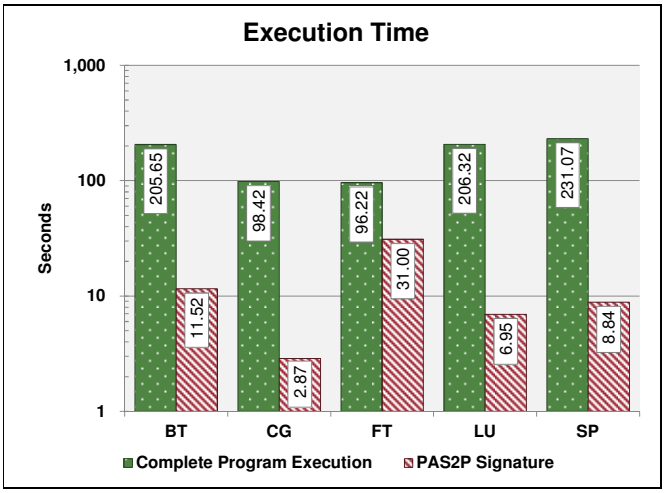

Figure 5. Execution time.

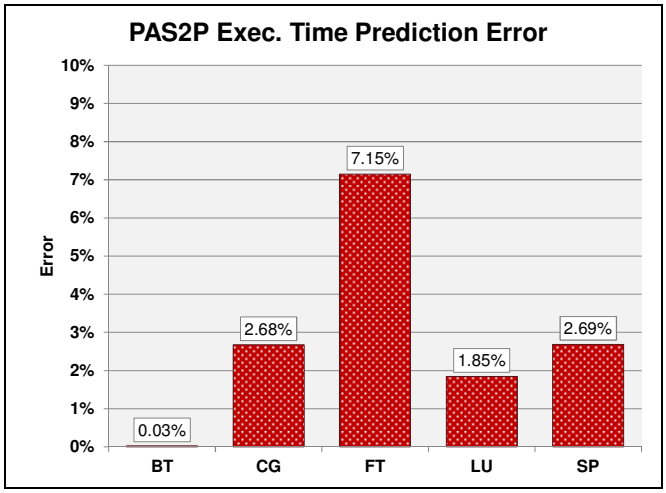

Figure 6. Error in PAS2P execution time prediction.

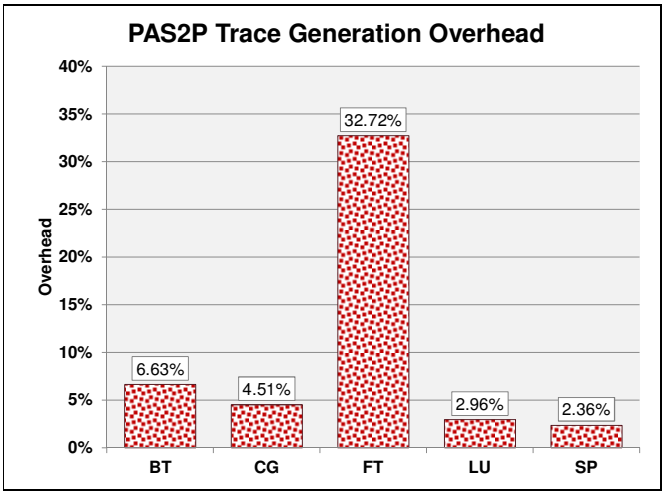

Figure 7. PAS2P trace generation overhead.

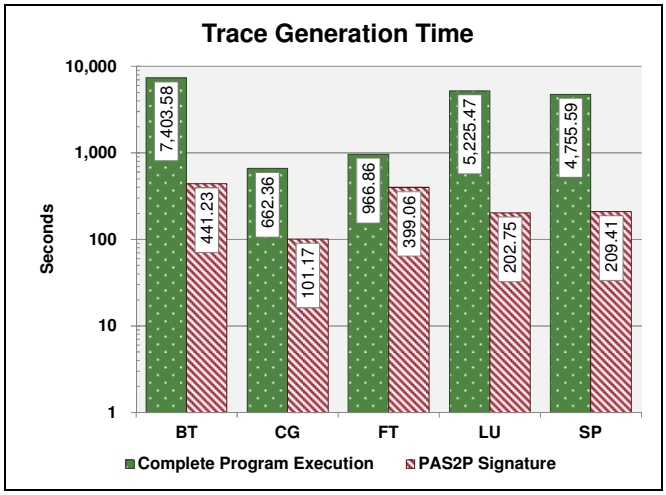

Figure 8. Trace generation time.

So, PAS2P had to execute proportionally more instructions of the program and achieved a worse prediction of the **FT** benchmark execution time. As we will show in section V.C, this is not a problem to the robustness prediction.

B. Basic block trace generation and robustness evaluation

The second step of our experimental work consisted of, once obtained the PAS2P signature, generating the basic block trace of the complete programs executions and of the PAS2P signature executions.

Fig. 8 shows that the trace generations of the PAS2P signatures were considerably faster than the trace generations of the whole programs execution.

Even spending less time by tracing the PAS2P signatures than tracing the whole programs, the size of the basic block traces (in bytes) shown in Fig. 9 presents a reduction of less than 10% on average.

This occurs because of:

a) Our basic block tracing tool compresses the traced data based on basic block sequence repetitions on the fly during the trace generation (not after the trace is generated);

b) In the experiments for this work, the portion of the trace file that contains the basic block sequences (more influenced by PAS2P) represented on average 22.9% of the whole trace file, meanwhile the portion of the trace file with the information about the architecture instructions in the basic blocks (less influenced by PAS2P) represented 77.1%.

In Fig. 10 we present the time needed to calculate the robustness of the evaluated programs and its respective PAS2P signatures.

The **LU** benchmark achieved the best gain in time saving of the robustness analysis. It needed only 3.57% of the time spent by the standard program analysis to complete its work.

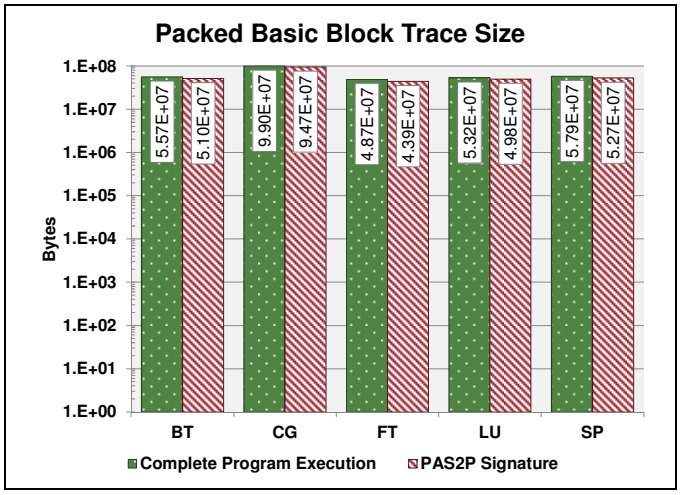

Figure 9. Packed basic block trace size.

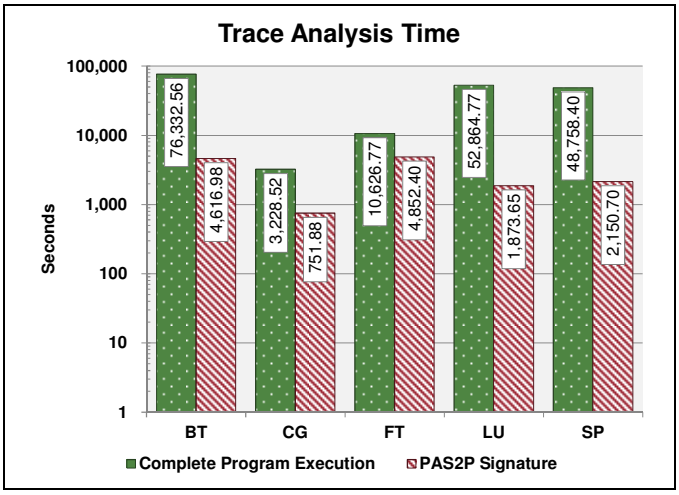

Figure 10. Trace analysis time.

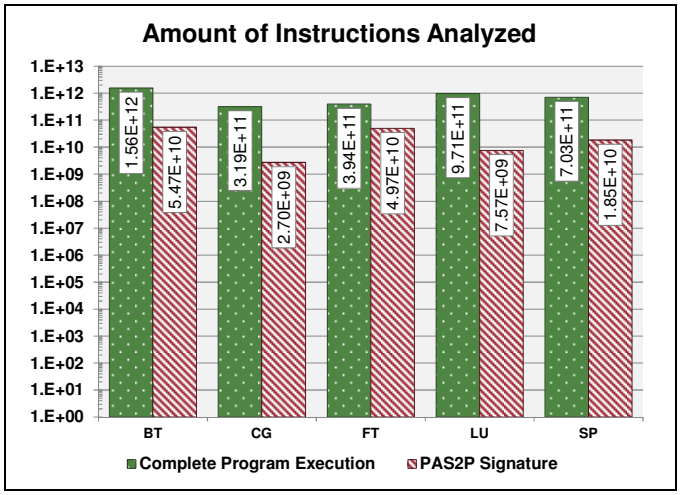

Figure 11. Amount of analyzed instructions.

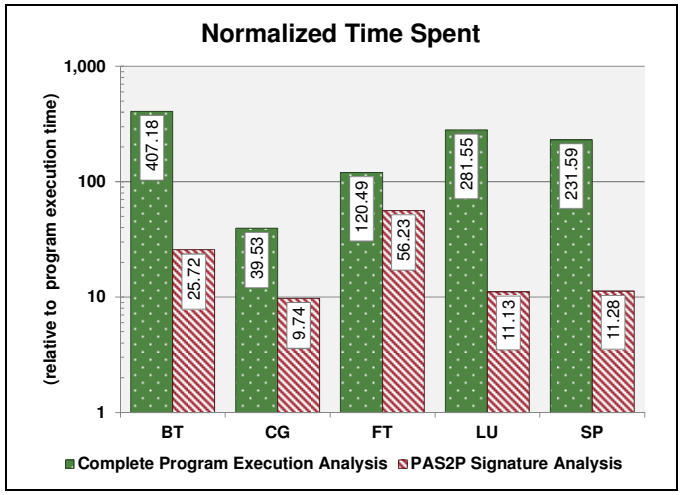

Figure 12. Normalized time spent.

The worst case, as we could foresee, was achieved by the **FT** benchmark (45.3%).

On average, the traces analysis of the PAS2P signatures needed 16.25% of the time required to analyze the whole programs traces (8.99% without the **FT** benchmark).

As we previously mentioned, the basic block trace size of the PAS2P signatures weren't significantly smaller than the whole programs basic block traces. However, the amount of instructions analyzed in each case of robustness analysis for the PAS2P signatures were significantly smaller (Fig. 11).

The robustness analysis of the PAS2P signatures needed, on average, 4.07% (1.94% without the **FT** benchmark) of the instructions of the whole program analysis to accomplish its work.

In Fig. 12 we compare the time needed to perform the whole analysis of the programs' robustness: with and without PAS2P.

The PAS2P Signature Analysis time takes into account not only the robustness trace and the analysis, but also the PAS2P signature generation time and the PAS2P trace analysis time.

While the whole program robustness analysis needed 216 times the programs execution time (on average), using PAS2P and evaluating the robustness of the PAS2P signatures needed 23 times the programs execution time (14 without the **FT** benchmark).

C. Comparing evaluated and predicted robustness's

After evaluating the time needed to calculate the robustness's we compared the results of the calculated robustness (Fig. 13) and the robustness prediction of the PAS2P signatures (Fig. 14).

The results obtained with the prediction of the PAS2P signatures' robustness were very accurate in comparison with the numbers obtained for the complete programs' executions.

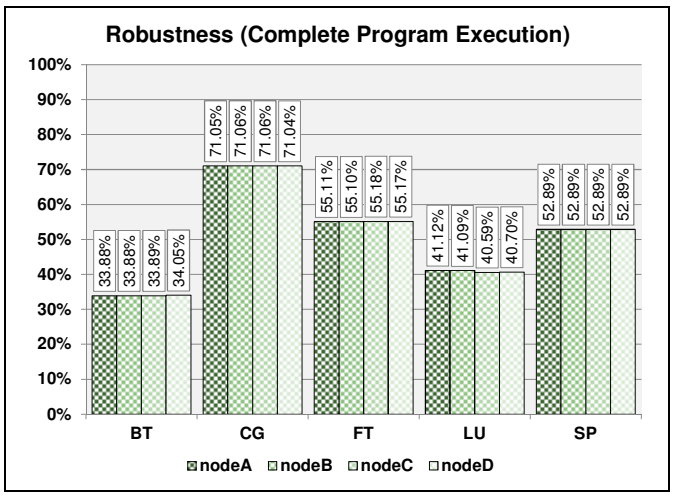

Figure 13. Robustness of the complete program execution.

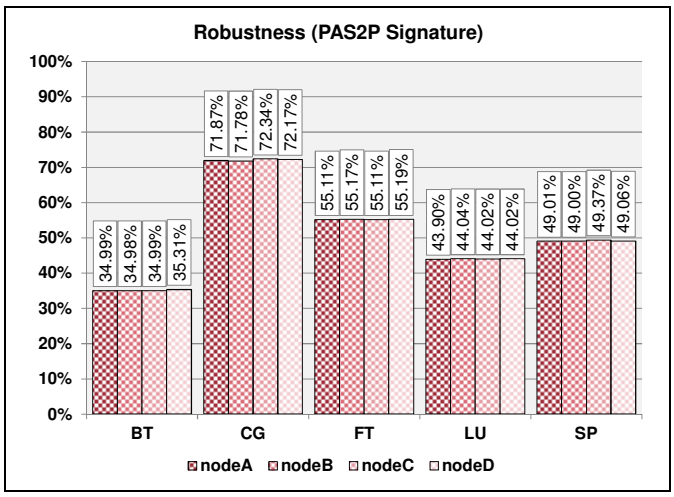

Figure 14. Robustness of the PAS2P signature execution.

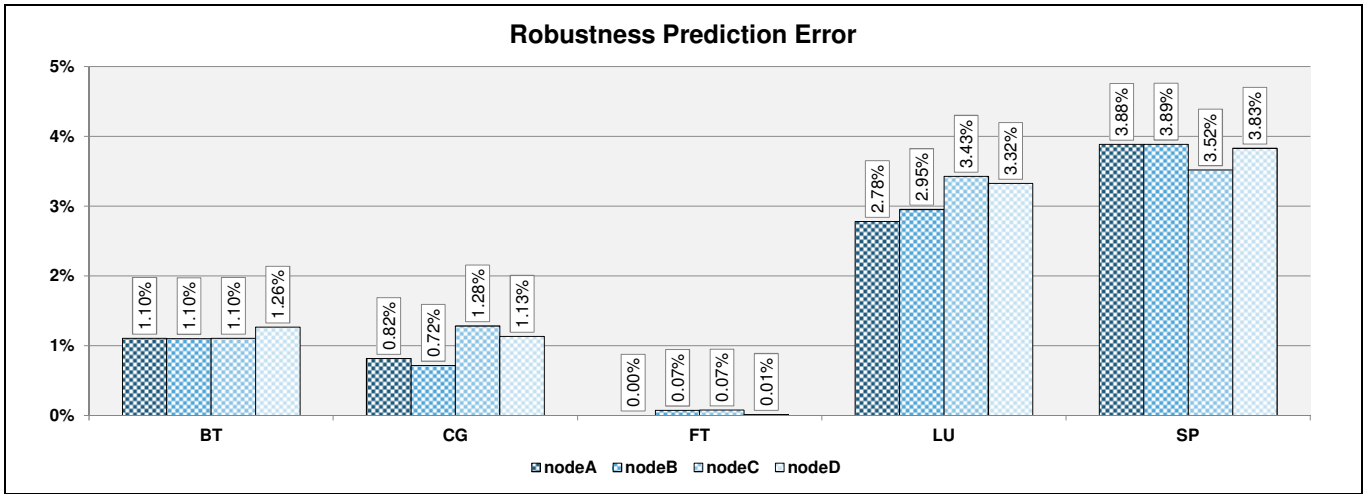

Figure 15. Robustness prediction error.

All the PAS2P signatures robustness's predictions achieved an error lower than 4% as show in Fig. 15.

As the **FT** benchmark was the program that needed more execution than the others when running its PAS2P signature, its robustness evaluation took into account more real information and needed less extrapolation, scoring the best robustness prediction of this set of experiments with an error lower than 0.1%.

VI. CONCLUSION AND FUTURE WORK

Evaluating a program's behavior in presence of transient faults by using fault injection environments is often a very time consuming work.

In this paper we combined our previously proposed methodology of calculating a program's robustness against transient faults with PAS2P, a methodology to estimate a program execution time based on a small fraction of it (its signature). This combination allowed us to predict the robustness against transient faults of parallel programs based on MPI.

The new proposed method estimated the amount of unACE bits of all parallel programs processes by analyzing the execution trace of its signatures. We were able to estimate the robustness almost 20 times faster on average than our previous approach and achieving an error less than 4% on average in the predicted robustness.

PAS2P is more effective the more repetitions the evaluated program has (needing a smaller signature to the prediction). This increases the scale of problems that could be tested. The combination of both methodologies will allow us to study the robustness of very large parallel programs by only analyzing its PAS2P signature.

One next step of this work is to improve even more our robustness calculation tool, making it faster by saving computing time taking into account the repetition of program basic blocks during its execution over a given architecture.

As in the current methodology we only classify a program unACE bits, in the next step of our work we will classify the ACE bits in two categories: DUE and SDC. We will improve even more our robustness evaluation by knowing precisely the amount of DUE bits of a program.

REFERENCES

[1] N. J. Wang, J. Quek, T. M. Rafacz, S. J. Patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," in Proceedings of the 2004 International Conference on Dependable Systems and Networks, pp. 61—70.

[2] R. Baumann, "Soft errors in advanced computer systems" in Design & Test of Computers, 2005, vol. 22, pp. 258—266.

[3] J. Gramacho, D. Rexachs, E. Luque. "A Methodology to Calculate a Program's Robustness against Transient Faults" in Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp. 645—651.

[4] A. Wong, D. Rexachs, E. Luque. "Extraction of Parallel Application Signatures for Performance Prediction" in Proceedings of 12th IEEE International Conference on High Performance Computing and Communications (HPCC), 2010, pp. 223—230.

[5] Shubhendu S. Mukherjee, Joel Emer, Steven K. Reinhardt. "The Soft Error Problem: An Architectural Perspective," in Proceedings of the 11th International Symposium on High-Performance Computer Architecture, Washington, DC, 2005, pp. 243—247.

[6] B. Nicolescu, R. Velazco, "Detecting soft errors by a purely software approach: method, tools and experimental results," in Design, Automation and Test in Europe Conference and Exhibition, 2003, pp. 57—62.

[7] N. Oh, P. Shirvani, E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," in IEEE Transactions on Reliability, 2002, vol. 51, pp. 63—75.

[8] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, S. S. Mukherjee, "Software-controlled fault tolerance," in ACM Transactions on Architecture and Code Optimization, 2005, vol. 2, pp. 366—396.

[9] G. A. Reis, J. Chang, D. I. August, R. Cohn, S. S. Mukherjee, "Configurable Transient Fault Detection via Dynamic Binary Translation," in Proceedings of the 2nd Workshop on Architectural Reliability (2006).

[10] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," Programming Language Design and Implementation (PLDI), Chicago, IL, June 2005, pp. 190—200.