PROMON: a profile monitor of software applications

DDECS
DESIGN & DIAGNOSTICS of ELECTRONIC CIRCUITS & SYSTEMS

8th IEEE Workshop on
Design and Diagnostics of Electronic Circuits and Systems
Sopron, Hungary, April 13 - 16, 2005

PROMON: A PROFILE MONITOR OF SOFTWARE APPLICATIONS

A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri, C. Tibaldi
Politecnico di Torino, Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24, 10129, Torino (Italy)
{alfredo.benso, stefano.dicarlo, giorgio.dinatale, paolo.prinetto, luca.tagliaferri, clara.tibaldi}@polito.it

Abstract. *Software techniques can be efficiently used to increase the dependability of safety-critical applications. Many approaches are based on information redundancy to prevent data and code corruption during the software execution. This paper presents PROMON, a C++ library that exploits a new methodology based on the concept of "Programming by Contract" to detect system malfunctions. Resorting to assertions, pre- and post-conditions, and marginal programmer interventions, PROMON-based applications can reach high level of dependability.*

1 Introduction

With the recent progress of the technology it is simple to design and produce high performance low cost microprocessor systems. Unfortunately, these systems do not usually guarantee high levels of reliability because of mainly two reasons: to lower as much as possible the Time-to-Market they are designed without the necessary care to dependability issues; new and smaller technologies are much more sensitive than older systems to environmental perturbations that cause transient errors [1][2].

Recently, Commercial Off-The Shelf (COTS) hardware and software components have been introduced in safety-critical automotive and space applications; these components guarantee high performances at the price of a low dependability [3]. COTS hardware cannot be modified. The only possible low-cost solution is to take advantage of SIHFT (Software Implemented Hardware Fault Tolerance) techniques that allow, by using software elements only, to detect and correct errors [4][5]. SIHFT techniques are, in general, based on the addition, to the original target application, of software routines able to check the validity and correctness of the executed code and the managed data.

This paper presents PROMON, a software able to intercept faults that may occur during the execution of the application and block the execution; the innovative aspect of the approach is that the implemented methodology relies on the principle of Programming by contract, and on the use of common assertions, pre and post-conditions, to ensure that the code follows its expected behavior [6][7][8][9].

Many techniques have been proposed in the past in order to improve the reliability of non-reliable COTS; the majority of them are based on the insertion of redundant code and data in the running applications, so that each single operation is entrusted by at least one re-computation. The redundant information can be inserted statically in the original high level code during a preliminary phase: this is the case of source-to-source compilers like Porch

[10] and Recco [11]. Other approaches as BSSC, ECI and watchdogs are designed to check the consistency of the executed machine code [12]. Watchdogs stem from the idea that errors (software or hardware) bring the processor to follow an execution flow different from the expected one. A Watchdog is usually a hardware/software component able to check at runtime the correctness of the execution flow [13].

The paper is organized as follows: PROMON architecture is sketched in Section 2; Section 3 proposes some experiments to show PROMON performances. Section 4 presents the conclusions and outlines future works.

2 PROMON Architecture

PROMON implements a pure software fault-tolerance technique based on assertions. Transient errors can cause different types of system errors such as the corruption of a memory location that stores a variable. This type of errors can easily propagate to other variables. PROMON offers a methodology to observe the value of the variables during the application execution and to detect if the value of each "protected" variable is legal or not.

PROMON is implemented as a C++ [14] template library based on the use of assertions (pre and post-conditions) on the application variables. Pre and post-conditions can be manually setup by an expert programmer or can be automatically extracted from the analysis of the behavior of each variable during a set of fault-free executions of the application. Pre and post conditions are extracted in the three phases (Figure 1) summarized here and detailed in the next sub-sections:

1. *Code instrumentation*: a set of selected critical variables are "redefined" so that, during the software execution, their behavior can be traced on a log file. Techniques to select the most critical variables have been presented in [15];
2. *Golden executions*: the user runs the application with different workloads. During each execution the value of each protected variable is logged for later analysis. The choice of the workloads to apply is a very critical task. In general, the greater the workload is, the more precise the extracted assertions will be and less probable to generate "false negatives" (errors not recognized as such);
3. *Assertion extraction*: all the log files are analyzed and, for each variable, a set of assertions is extracted. In this phase users can also provide a set of "user-defined" assertions, maybe difficult to extract but known as true by the programmer.

At this point, the value of each protected variable is validated, at run-time, using the extracted assertions; if PROMON finds any kind of violation an exception is raised and the execution is stopped.

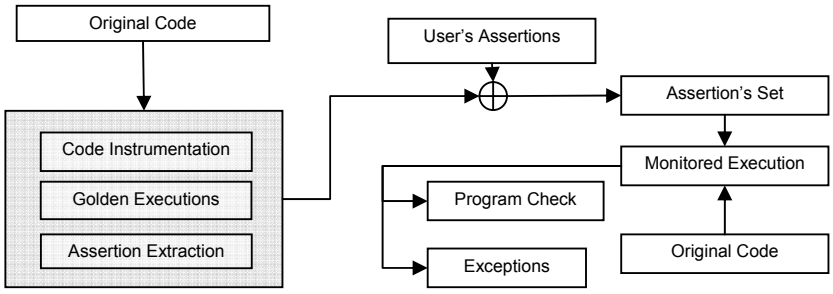

Figure 1: PROMON Architecture

2.1 Instrumentation

The only manual intervention required by the proposed methodology is the instrumentation phase, where the original definition of the variables that need to be protected has to be replaced with the predefined types included in the PROMON C++ library (PROMON<int>, PROMON<char>, PROMON<float>, …). The set of possible pre and post-conditions that can be specified and that PROMON is able to extract are: <Greater then *n* / Greater than *variable*>, < Lower than *n* / Lower than *variable* >, <Even / Odd>, <Always increasing / Always decreasing>, <Multiple of *n* / Multiple of *variable*>, <Is the index in a cycle>, <User-defined set condition>. In addition, the programmer has the possibility of specifying for each variable a set of valid ranges or validation functions.

Apart from the variables' redefinition the user is not expected to change anything else in the code since a complete set of overloaded operators let protected variables behave exactly as normal type variables. Therefore, each original operation is executed as planned but an additional routine is executed to log the behavior of each variable. An example of redefinition of the "=" and the "++" operators is presented in Figure 2.

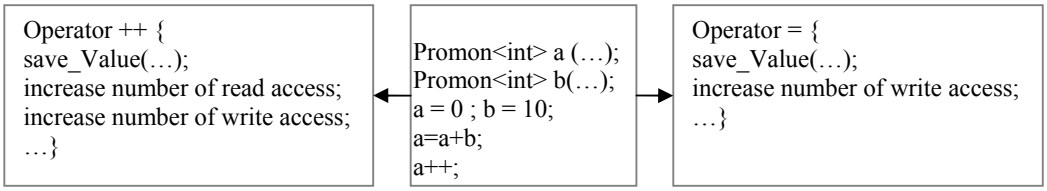

Figure 2: PROMON operator overloading

The resulting program is functionally equivalent to the original one except from the fact that during the execution the program itself logs significant variable usage statistics, such as the number of times a variable is written and read, and each value the variable stored during the execution.

2.2 Golden execution and assertion extraction

This phase is probably the most important one, since the choice of the workloads directly affects the "precision" of the assertions PROMON is able to extract. Applying different workloads allows logging a number of "golden" (surely correct) executions of the target application: PROMON analyzes these executions to derive a list of valid assertions for each variable. It is programmer's duty to choose a sufficient number of golden runs to build a significant application profile.

The analysis of the golden logs then tries to map each variable's behavior in one or more of the pre and post-conditions listed in Table 1. As an example, consider a variable x that during three different golden executions assumes the values listed in Table 2.

1st execution: 2,6,10	2nd execution: 4,8,12	3rd execution: 10,14,18

Table 1: Example of assertion extraction

From the analysis of the logged values PROMON is able to extract the following assertions for variable x: *Greater than 2, Even, Always increasing, Lower than 18*

At the end of the analysis of the golden executions a "readable" text file is produced with all the extracted assertions.

2.3 On-line execution

After the assertions extraction, the behavior of the protected variables changes. Their task is not anymore to log the variable behavior but to check that the value of a protected variable is compatible with the set of assertions for that variable. The behavior of the on-line check is summarized in Figure 3.

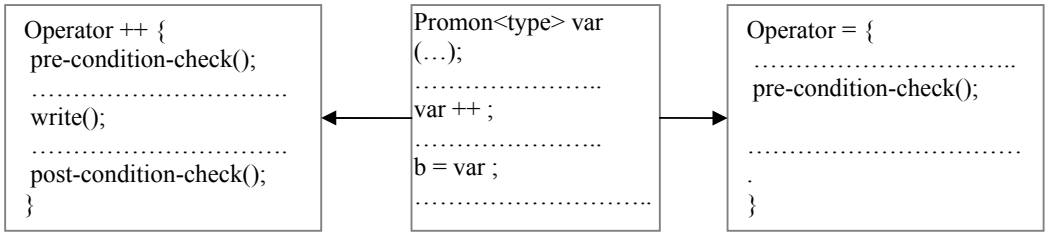

Figure 3: Condition check

If, during execution, all the pre and post-conditions are satisfied, the program terminates normally; otherwise, in case of an error detection, PROMON can proceed (as the user prefers) in two ways: in "*Exception Mode*" an exception routine is called and the application execution is interrupted; in "*Warning Mode*" the execution continues and, when the application ends, the presence of anomalies is signaled to the user via a log file containing the list of the violated assertions.

2.4 Self injection

In order to evaluate the effectiveness of the proposed approach, a set of Fault Injection experiments has been executed. In particular a number of faults have been artificially injected inside the application variables in order to check if the extracted assertions where precise enough to eventually detect an error in one or more application variable.

The PROMON C++ library can be configured to perform random injections in the protected variables during the application execution. Figure 4 shows how the injections and errors detection flow are performed.

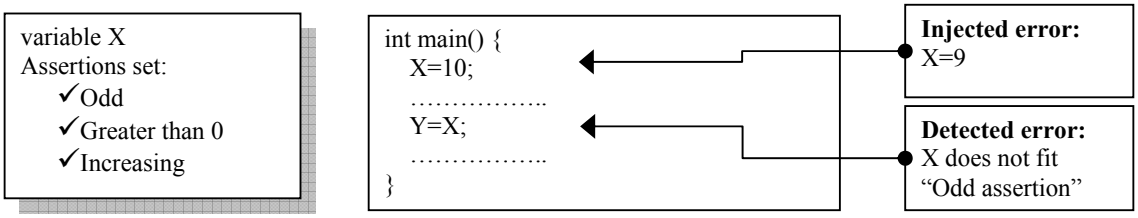

Figure 4: PROMON self injection

At the end of a self-injected execution PROMON provides a log file containing the injected variable, the correct and wrong values detected in each variable, the output of the application (to detect if the results are correct or not), and the detected errors.

3 Benchmark and experimental results

This Section presents the experimental results obtained applying PROMON to the following set of benchmark applications: Whetstone [16] (a CPU/RAM benchmark), Stream [17] (a RAMtoRAM transfer-rate calculator), a Knapsack problem solver [17] and a Dhrystone application [16]. We executed each application 1,000 times, each time injecting a random fault in one of the application variables. In the following experiments, where not mentioned otherwise, all variables of the application have been protected by PROMON.

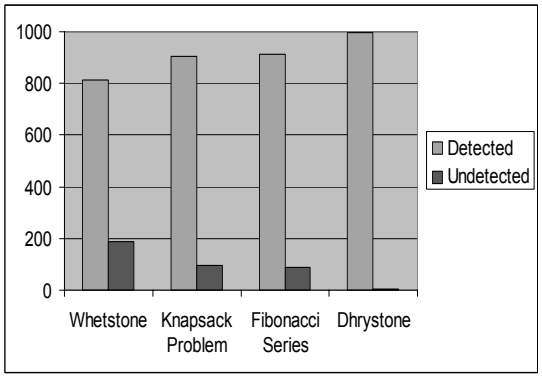

Figure 5: PROMON error detection capability

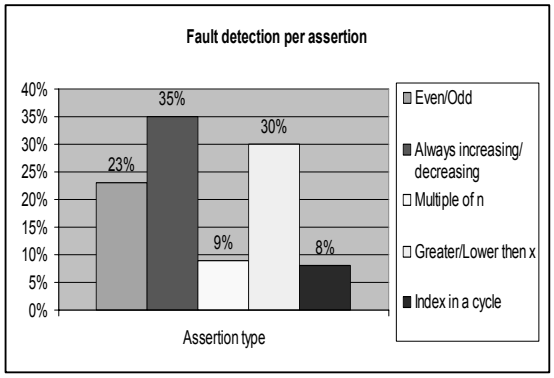

Figure 6: Fault detection per assertion

As it can be seen from Figure 5, the detected errors vary from 80 to 99% on an overall number of one thousand injections for each tested application. The different detection rates is also due to the different behavior of the variables in each benchmark: the more a variable is read, the greater is the influence on other variable, and therefore the probability that the error is spread and, eventually, an assertion will fail. Figure 6 shows the detection capabilities of each type of assertion and, as it can be seen, not all the assertions have the same performance. The reason why the sum of the percentage is greater than 100% is that for a number of variable errors are detected by more than one assertion.

Finally a set of user-defined assertions has been introduced in the four benchmarks; the new assertions have been designed by an expert programmer analyzing the source code: these user-defined assertions increase the average error detection by a 5%.

PROMON assertions, obviously, introduce a time overhead during the application execution: this extra time depends on the number of monitored variables and the number of active assertions. Figure 7 and Figure 8 show the execution time overhead for the Dhrystone application.

Each assertion give a different contribution to time overhead: the two heaviest assertions are the ones pertaining bounds and parity check; this event is caused by the fact that these are the most commonly applicable constraints for variables.

The number of variables influences the time overhead: for the Stream benchmark, which only has four protected variables, the increase in execution time is about 3%.

The memory overhead instead is constant for all the benchmarks (about 1 KB); this is due to the fact that for middle size applications PROMON memory allocation is almost independent from the number of assertion and variables monitored.

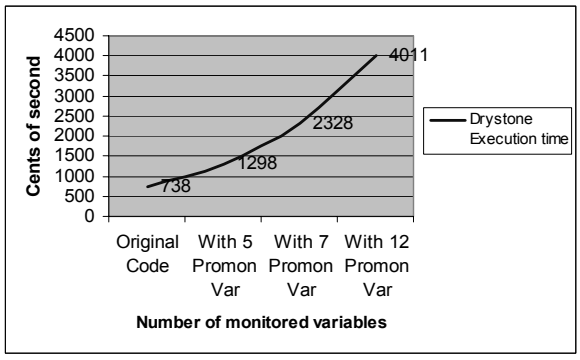

Figure 7: Time overhead w.r.t. the number of monitored variables

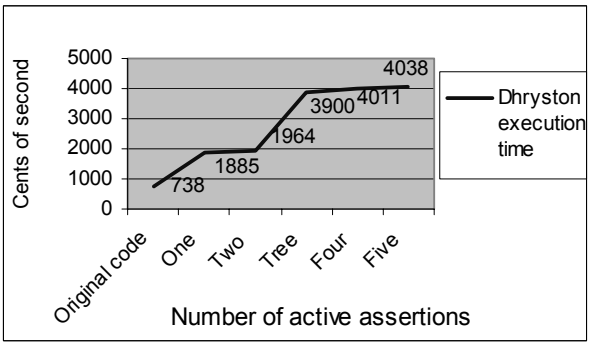

Figure 8: Time overhead w.r.t. the number of active assertions

4 Conclusion

This paper proposes a technique that shows how some "Programming by Contract" paradigms can provide a new approach to Software Implemented Hardware Fault Tolerance. The use of pre and post-conditions, automatically extracted and verified at run-time, seems to be an excellent alternative to traditional information redundancy techniques. The main advantage of the proposed approach is that it requires a very low manual intervention still guaranteeing very significant error detection rates. Moreover, programmers can choose to use the assertion mechanism for a reduced and critical subset of variables, so to trade-off between CPU effort and detection capabilities. The experiments show that the proposed technique results in a significant increase of the benchmarks reliability; on the other hand the time overhead introduced by the extra computation can increase rapidly depending on the number of the monitored variables.

Future work will investigate the possibility of creating dependencies between assertions in order to make the validation mechanism more precise and fast.

References

[1] F. Faccio, C. Detcheverry, M. Huhtinen CERN, Geneva, Switzerland,"First evaluation of the Single Event Upset (SEU) risk for electronics in the CMS Experiment", CMS NOTE 1998/054 CERN, Geneva, Switzerland.

[2] P.P. Shirvani, N.R. Saxena, E.J. McCluskey, "Software-implemented EDAC protection against SEUs", IEEE Transactions on Reliability, vol. 49 Issue: 3, Sep 2000, Page(s): 273 -284.

[3] P.P. Shirvani, N. Oh, E.J. McCluskey, D.L. Wood, M.N. Lovellette, K.S. Wood, "Software-Implemented Hardware Fault Tolerance Experiments: COTS in Space" International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8), New York (NY), 2000, Page(s) B56-57.

[4] D. Todd Smith, T. A DeLong, B. W. Johnson, J. A. Profeta III, "An Algorithm Based Fault Tolerant Technique for Safety Critical Applications", Reliability and Maintainability Symposium, Philadelphia, 1997.

[5] M. Zenha Rela, H. Madeira, J. G. Silva, "Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks", 26th International Symposium on Fault-Tolerant Computing (FTCS-26), Sendai (J), 1996, Page(s). 394-403.

[6] Rosenblum, D.S., "A practical approach to programming with assertions"; IEEE Transactions on Software Engineering , Volume: 21 , Issue: 1 , Jan. 1995 Pages:19 – 31.

[7] J.M Voas , K.W. Miller, "Putting assertions in their place"; Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on , 6-9 Nov. 1994 Pages:152 – 157.

[8] Yin Hwei, J.M. Bieman, "Improving software testability with assertion insertion", Test Conference, 1994. Proceedings., International , 2-6 Oct. 1994. Pages:831 – 839.

[9] Meyer B., "Applying Design by Contract", IEEE Computer, Volume: 25 , Issue: 10 , Oct. 1992 Pages:40 - 51

[10] V. Strumpen, Portable and Fault-Tolerant Software Systems, IEEE Micro, September-October 1998, pp. 22-32

[11] Benso, A.; Chiusano, S.; Prinetto, P.; Tagliaferri, L.; "A C/C++ source-to-source compiler for dependable applications"; Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on, 25-28 June 2000 Pages:71-78.

[12] Miremadi, G., Johan Karlsson, Ulf Gunneflo and Jan Torin. Two software Tecniques for On-line Error Detection. Digest of Papers, 22nd Annual International.

[13] D.J. Lu, "Watchdog Processor and Structural Integrity Checking", IEEE Transactions on Computers, vol. C-31, No. 7, pp. 681-685, July 1982.

[14] The C++ Programming Language (Third Edition and Special Edition) Addison-Wesley.

[15] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri, Data Criticality Estimation in Software Applications, IEEE International Test Conference, Charlotte (NC), October 2003, pp. 802-810

[16] W.J. Price, "A benchmark tutorial"; Micro, IEEE , Volume: 9 , Issue: 5 , Oct. 1989 Pages:28 – 43.

[17] http://www.cs.virginia.edu/stream/ref.html[21] R. Hinterding, "Representation, constraint satisfaction and the knapsack problem" , Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on, Volume: 2 , 6-9 July 1999 Pages: 1286-1292 Vol. 2.