Control-flow checking via regular expressions

Control-Flow Checking Via Regular Expressions

ALFREDO BENSO, STEFANO DI CARLO, GIORGIO DI NATALE, PAOLO PRINETTO, LUCA TAGLIAFERRI
Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24 - I-10129, Torino, Italy
Email: { benso, dicarlo, dinatale, prinetto, tagliaferri }@polito.it
http://www.testgroup.polito.it

Abstract

The present paper explains a new approach to program control-flow checking. The check has inserted at source-code level using a signature methodology based on regular expressions. The signature checking is performed without dedicated watchdog processor but resorting to inter-process communication (IPC) facilities offered by most of the modern Operating Systems. The proposed approach allows very low memory overhead and trade-off between fault latency and program execution time overhead.

1. Introduction

The use of computer-based systems pervades all areas of our lives from common house land applications such as microwave ovens and washing machines, to complex applications like aircraft, trains and medical control systems, allowing high productivity and flexibility. They are commonly referred to as *"Embedded Computer Systems" (ECS).*

The proposed examples state a very large number of ECS play a key role in critical tasks with respect to human safety and data security, supported by the development of new and powerful electronics circuits. In this context, high availability is a must to guarantee human safety.

As devices geometry decreases, circuits clock frequencies increases and processors are introduced into more electrically active environments; the incidence of transient errors increases, decreasing the dependability of ECS where these components are used. These transient errors can only be detected by concurrent error detection techniques, allowing the maintenance of acceptable levels of system dependability.

The design of ECS is always constrained by the reduction of time-to-market. This makes not feasible the development of custom products with high performances and dependability levels. This requirement forces the engineer to systematically use commercial off-the-shelf components in both software and hardware domain. These components are normally not designed to work in stressed environments and do not guarantee high dependability levels. The challenge is to build fault tolerant systems that harness the market forces by using off-the-shelf hardware and software components.

Classical approaches for dependable ECS development rely on hardware redundancy. Although they are effective in protecting against transient faults, they are usually expensive. A possible and less expensive approach is to move the problem of fault detection at software level using software redundancy techniques.

Control-flow checking has become a widely studied approach to concurrent error checking. The on-line test is aimed at detecting erroneous sequences of instructions in a program execution [1] [2]. The proposed solutions mainly rely on dedicated hardware [3]. The basic schema is based on watchdog and signature analysis [4] [5] [6] [7] [8] [9] [10]. The application program is split into elementary blocks with single entry. A reference signature representing the correct execution flow in the blocks is calculated off line and stored. At run time the signature is calculated again and compared with the golden one using so called dedicated *watchdog processors.* These hardware techniques have been evaluated on medium size applications using different hardware platforms [11].

The main difference in the various approaches is in the method used to calculate and check the signature. In particular it is possible to identify two classes of approach:

- *Embedded Signature Monitoring (ESM)*: where the signature is embedded in the application program [5] [6];
- *Autonomous Signature Monitoring (ASM):* where the signature is stored in a dedicated watchdog memory [8].

Despite the effectiveness of the proposed methodologies they usually need dedicated hardware and system modifications, in contrast with the constraints of using

1081-7735/01 $10.00 © 2001 IEEE

commercial off-the-shelf components. In addition, with the increasing size of processor cache memory, the detection capability of these techniques decrease since they are able to detect faults into main memory, accessible by the watchdog processor but not faults inside the processor cache.

In order to solve the problem of hardware control-flow checking, some software approaches have been proposed. They rely on the insertion of appropriate instructions into the code to calculate and check the signature without needed a dedicated watchdog processor. Representative software based control-flow monitoring schemes are Block Signature Self Checking (BSSC) [12] and Control Checking With Assertion (CCA) [13].

The present paper proposes a new high-level approach to control flow based on multiprocess/multithread operating systems [14]. The target is the insertion of control-flow checking mechanism directly at source code level. The approach relies on the multiprocess/multithread programming facilities offered by most of the modern Operating Systems to minimize the modification of the target application and to allow trade off between time overhead and fault latency.

The proposed approach uses a new signature schema based on the *regular expression* formalism [15]. It has been implemented in a tool and evaluated using a custom fault injector [16].

The paper is organized as follow: Section 2 introduces some basic concepts and definitions related to the program flow theory. Section 3 proposes the new signature schema whereas Section 4 explains the control-flow checking methodology. To prove the effectiveness of the work Section 5 reports experimental results performed on a set of benchmarks. Finally Section 6 draws some conclusions.

2. Program Control Graph

Before presenting our control-flow checking methodology, we first introduce some definitions and models used in the sequel. A generic program can be represented by a so-called *Flow Control Graph (FCG)* in which each node represents an instruction, whereas the arcs represent the authorized sequences of instructions (Figure 1).

The nodes of a FCG can be grouped into two main classes:

- *Sequential Nodes:* the associated instruction does not modify the flow of the program, i.e. the program flow is sequential;
- *Control Nodes:* the associated instruction is able to modify the program flow. They are typically associated with flow control statements (*if, while, etc.*).

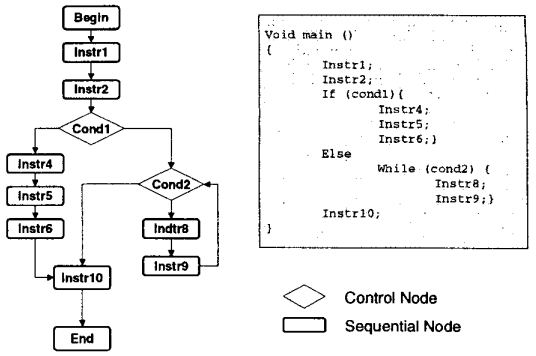

Figure 1: *Control Flow Graph*

Using the information provided by the FCG it is possible to split the program into *branch-free blocks.* A branch free block is a set of consecutive sequential vertexes in the FCG. A branch-free block is normally followed by a control vertex. Figure 2 shows the FCG of Figure 1 modified introducing the branch-free blocks. This modified FCG (MCFG) is the starting point for the proposed control-flow checking mechanism.

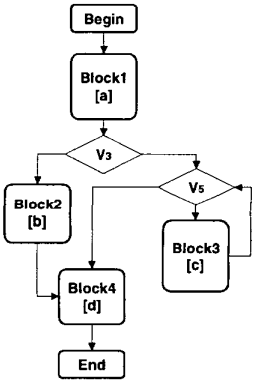

Figure 2: *Modified Control Flow Graph*

3. Flow Signature using Regular Expression

In the proposed approach the problem of control-flow checking is tackled resorting to a new signature approach able to identify all the allowed program flows executions. These program flows executions are identified by all the possible paths in the related MFCG starting from the

"Begin" node and ending in the "End" node (Figure 2). The defined signature is stored and checked at run time.

The novelty of our approach is in the use of *regular expressions* to calculate the program flow signature, instead of classical approaches based on LFSR or arithmetic functions.

Each branch-free block in the MFCG is labeled with a unique symbol named *block symbol* (in square brackets in Figure 2). Using this notation the allowed program flows executions generate a language composed by all the strings obtained by the concatenation of the block symbols composing the correspondent path in the MFCG. This set of strings can be considered as a language L in the sense of compiler theory [15]. This language can be represented in a formal way as:

$$L = (A, R)$$

Where:

- A is the *input alphabet* composed by all the defined block symbols;
- R is a *regular expression* able to generate all the necessary strings.

The language L can represent a signature for the target application program.

A program flow execution is allowed only if the corresponding path in the MFCG generates a string *(Control String)* belonging to the language L, in the other case a flow error has occurred.

Considering the example of Figure 2 the language L is:

$L = (A, R)$

Where

- $A = (a, b, c, d)$
- $R = a\,(b \mid (c)^*)\,d$

If an execution produce the control string S = *"acccd"* we can say that the execution belongs to the space of correct executions, whereas in case of the control string S= *"abcf"* a flow error has occurred since the string is not recognized by the language L.

The choice of using regular expressions for signature computation relies on the high expressivities of this formalism and the high simplicity in storing and manipulating this kind of strings. Using this formalism, the problem of control flow checking is reduced to the problem of generating the control strings during the execution of the application program and verifying for their correctness by checking whether it is accepted by the language L.

4. Control-Flow Checking

This paragraph explains our methodologies to generate and check control strings during the program execution. Both the code for strings generation and strings checking are obtained by modifying the original C/C++ source code of the target program application. This is not a limitation since the approach is general enough to work at any level of language: high-level, assembly-level and machine-level. The inserted code makes use of the multiprocess/multithread programming facilities provided by all the modern Operating Systems [14].

The application program and the checker program are instantiated as two different processes communicating using a pipe or any other Inter Process Communication (IPC) facilities. The checker process stores the language L associated to the application program and the code needed to check if control strings belong to the language. The application program is modified inserting at the end of each branch-free block the code needed to generate the related control symbol. This symbol is transmitted to the checker process by using the IPC (Figure 3). In case of a wrong symbol it can detect a flow error. The code needed to generate control symbols is normally very simple, like a write operation on a pipe or a call to an IPC function, thus maintaining the memory and time overhead very low. At the same time the implementation of the checker process is very simple since the task of checking the appartenence of a language using regular expression is very simple [15] having a very low impact on the final application.

The checker process is inherently equivalent to a finite state machine (FSM) that is usually not deterministic. Therefore, to be implemented it must be converted into a deterministic FSM, and this process may cause an exponential growth of states. In our approach the problem is implicitly solved by the method used to label the branch free blocks, that assure the generation of regular expressions recognized by deterministic FSMs.

The use of a multiprocess architecture allows a very powerful mechanism to trade-off between time overhead in the application program execution and target fault latency. This can be obtained by appropriately setting the priority of the two processes. At limit if only the final results are relevant and the occurrence of a fault during the program execution is not a risk for the target application, the checker process can be called only at the end of the program execution.

In addition, by splitting the branch-free blocks into sub-blocks and applying the same signature schema on the new MFCG obtained, it is possible to check also the flow of long sequences of sequential instructions, allowing a trade-off between dependability and time/memory overhead. The limit, in case of very critical applications, is to consider each instruction as a single branch-free block.

Figure 3 show the example of Figure 2 where the IPC is implemented as a pipe using the Linux System calls [17].

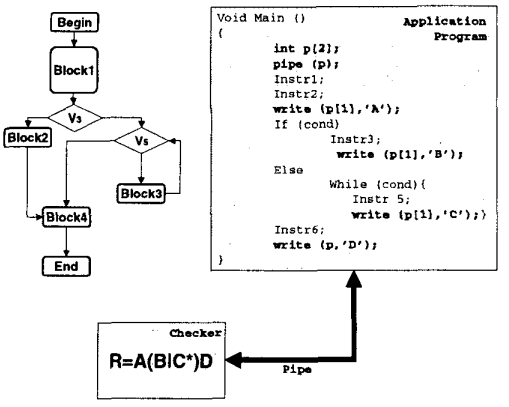

Figure 3: *Multi Process Checking Architecture*

5. Experimental Results

To evaluate the effectiveness of the proposed approach a source-to-source compiler has been implemented. It is able to build the FCG starting from a C/C++ source code and to identify the language L associated to the application program. It is therefore able to modify the original source code inserting the instructions needed to generate the block symbols, the source code of the checker and finally the instructions needed to make the proper connections between the two processes.

The compiler can deal with IPC mechanisms offered by both Microsoft Windows NT/2000 and Unix Operating Systems. The approach has been validated under Windows 2000 Operating System, using an ad-hoc fault injector [16] that allows injecting transient error into both data and code segments of an application program.

Experimental results have been gathered from five different benchmarks. For each benchmark, a preliminary analysis has been performed to evaluate the percentage of control-flow errors obtained during the injection experiments. The faults have been injected into the code segment of the target application.

Table 1 summarizes, for each benchmark and for both the original and modified source code:

- the binary code size;
- the execution time including, in the modified source code version, the communication time for the IPC mechanism and the time needed to check the correctness of the regular expression.;
- the number of crashes i.e. the number of faults injected that has generated a crash of the application program;
- the number of control-flow errors i.e. the number of injected faults that has caused a control flow different from the golden one ;
- the number of error not belonging in the set of control-flow errors;
- the number of detected control-flow errors, i.e. the number of injected faults belonging in the control flow errors category and detected by the control flow checking mechanism.

Table 1 shows that in general the proposed approach is able to sensibly reduce the incidence of control flow errors in the target application. The effectiveness of the approach is mainly influenced by the characteristics of the source code. Programs that made large use of branches and loops statements have major benefit from the control-flow checking whereas more sequential programs like the Floating Point benchmark are less influenced by the proposed strategy.

Moreover it is possible to note that the number of total detected errors is greater then the reduction of control-flow errors. This means that the proposed approach is also able to reduce the cases of crashes of the application program increasing the total dependability of the target application.

Concerning the memory overhead and time overhead Table 1 shows they are into an acceptable level considering that all the experiments have been performed in the worst case from the point of view of time overhead since the checker has been scheduled at each block symbol generation.

6. Conclusions

A very large number of computer based systems play a key role in critical tasks with respect to human safety and data security requiring high availability and dependability. The challenge is to build fault tolerant systems that harness the market forces by using off-the-shelf hardware and software components. Control-flow checking has become a widely studied approach to concurrent error checking.

In the present paper we presented a new high-level methodology to control-flow checking based on regular expressions and multiprocess/multithread Operating Systems. The main features of our approach are the possibility of working at different programming levels (high level language, assembly language, machine language), the low memory overhead and the high flexibility in terms of trade-off between time overhead and fault latency, and memory overhead and detection capability.

The proposed approach has been evaluated implementing a source-to-source compiler able to automatically insert the control structures. Experimental results demonstrate the effectiveness of the approach and the low overhead introduced in terms of both memory occupancy and execution time.

7. References

[1] S. S. Yau, F. Ch. Chen, *"An Approach to Concurrent Control Flow Checking"*, IEEE Transaction on Software Engineering, Vol. SE-6, No. 2, pp. 126-137, 1980.

[2] R. Leveugle, T. Michel, G.Saucier, *"Design of Microprocessors with Built-In On-Line test"*, 20th International Symposium on Fault-Tolerant Computing (FTCS-20), pp. 450-456, 1990.

[3] A. Mahamood, E. J. McCluskey, *"Concurrent Error Detection Using Watchdog Processor - A Survay"*, IEEE Transaction on Computer, Vol. 37, No. 2, pp. 160-174, 1988.

[4] M. Namjoo, *"Techniques for Concurrent Testing of VLSI Processor Operation"*, International Test Conference (ITC-82), pp. 461-468, 1982.

[5] M.A. Schutte, J.P. Shen, D. P. Siewiorek, Y. X. Zhu, *"Expermental Evaluation of Two Concurrent Error Detection Schemes"*, 16th International Symposium on Fault Toleran Computing (FTCS-16), pp. 138-143, 1986

[6] K. Wilken, J.P. Shen, *"Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Errors"*, IEEE Transaction on Computer Aided Design and Systems, Vol. 9, Issue 6, pp. 629-641, June 1990.

[7] H. Madeira, J. G. Silva, *"On-line Signature Learning and Checking"*, 2nd IFIP Working Conference On Dependable Computing for Critical Applications (DCCA-2), pp. 170-177, Feb. 1991

[8] T. Michel, R. Leveugle, G. Saucier, *"A New Approach to Control Flow Checking without Program Modification"*, 21th International Symposium on Fault-Tolerant Computing (FTCS-21), pp. 334-341, 1991.

[9] Shambhu Upaddhyaya, Bina Ramamurthy, *"Concurrent Process Monitoring with No Reference Signatures"*, IEEE Transaction on Computer, Vol. 43 no. 4, pp. 475-480, April 1994.

[10] G. Miremadi, J. Ohlsson, M. Rimen, J. Karlsson, *"Use of Time and Address Signatures for Control Flow Checking"*, 5th IFIP Working Conference on Dependable Computing for Critical Applcation (DCCA-5), pp. 113-124, 1995

[11] X. Delord, G.Saucier, *"Control Flow in Pipelined RISC Microprocessor: The Motorola MC88100 Case Study"*, Workshop on Real Time (Euromicro '90), pp. 162-169, 1990.

[12] G. Miremadi, J. Karlsson, U. Gunneflo, J. Torin, *"Two software techniques for on-line error detection"*, 22th International Symposium on Fault-Tolerant Computing (FTCS-22), pp. 328-335, July, 1992.

[13] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, *"Design and Evaluation of System-Level Checks for on-line Control Flow Error Detection"*, IEEE Transaction on Parallel and Distributed Systems, Vol. 10, No. 6, pp. 627-641, June 1999.

[14] Abraham Silberschatz, Peter Galvin, *"Operating System Concepts, 5th Edition"*, John Wiley & Sons, January 1998.

[15] A.V. Aho, R. Sethi, J. D. Ullman, *"Compilers: Principles, Techniques and Tools"*, Addison-Wesley, 1986.

[16] A. Baldini, A. Benso, S. Chiusano, P. Prinetto, *"BOND: An Interposition Agents based Fault Injector for Windows NT"*, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'2000), pp. 387-395, October 2000.

[17] Official Linux Web Site: http://www.linux.org

	Floating Point Benchmark		Matrix Multiplication Benchmark		Quick Sort		BubbleSort		DicotomicSearch	
	Original	Modified	Original	Modified	Original	Modified	Original	Modified	Original	Modified
Binary Code Size (KB)	48	53	14	17	15	18	36	40	36	39
Eexecution Time (s)	1,5	1,8	6,1	10,5	0,5	0,6	0,1	0,3	0,4	0,6
# Crashes	484	472	420	415	440	433	463	458	452	445
# Control Flow Errors	25	19	13	6	19	9	16	3	18	9
# Other Errors	26	26	30	30	29	29	33	33	29	29
# Detected Flow Errors		18		12		17		18		16

Table 1: *Experimental results*