Embedded Signature Insertion Based on Profiling Deployed Software Technique

Fabian Vargas, Cláudia A. Rocha, Antônio A. de Alecrim, Carlos A. Becker

Electrical Engineering Dept.
Catholic University – PUCRS
Av. Ipiranga, 6681.
90619-900 Porto Alegre – Brazil
vargas@computer.org

ABSTRACT

We propose a new approach[1], namely Optimized Embedded Signature Monitoring (OESM) to perform on-line control-flow fault detection. The underlined advantage of this approach is the ability to perform a profiling algorithm that analyses the control-flow graph of user programs in order to optimize the number of checkpoints (i.e., signatures) to be inserted along with the application code during the software development phase. By optimization, we mean to find, for a given application, the best trade-off between the minimum number of signatures for the maximum fault detection coverage. The embedded signatures are checked at runtime by the processor against compilation-time pre-computed values every time the processor reaches these signature points. In order to evaluate the validity of the proposed approach, experimental results have been carried out. The obtained results indicate that OESM minimizes the number of signatures embedded in the application code (thus, minimizing memory overhead and performance degradation) while maintaining the same (or at least a similar) fault-detection capability level when applied to conventional control-flow fault detection approaches. At present, a tool that partially automates the proposed approach is under development.

1. INTRODUCTION

For real-time embedded applications, system robustness is frequently achieved by means of *hardware redundancy* [1]. However, this solution results inevitably in more expensive systems. Additionally, power consumption and volume/weight also increase beyond affordable values for most of the mass-consumer products. An alternative solution to this trade-off is the use of *software redundancy* [2]. This option is much more cost-effective. However, it implies in system performance degradation and memory overhead that can rarely be accepted by applications devoted to real-time responses. These penalties are due to the extra checking instructions that must be inserted in the application code, at specific checking points. These instructions must be executed at runtime by the processor.

It is in this scenario that we propose a new technique, which privileges robustness and performance at the same time, while maintaining overall system cost at reasonable values. The technique is based on a profiling algorithm that analyses the control-flow graph of user programs to optimize the number of checkpoints to be inserted along with the application code during the software development phase. By optimization, we mean to find the best trade-off between the *minimum number of checkpoints* for the *maximum fault detection coverage*.

The remainder of this paper is divided as follows: *Section 2* describes the proposed approach, while *Section 3* presents the final conclusions of this work.

2. THE PROPOSED APPROACH

2.1. The Graph and the Adjacency Matrix

Consider a weighted graph *G(V,E)*, where *V* is the *vertices* (or node) and *E* is the *edge* that connects a pair of nodes. Consider also that $W_{i/j}$ is the weight of the edge connecting nodes V_i and V_j. Then, for the graph given in Fig. 1, V = {1,2,3,4,5,6,7} and E = {$100_{1/1}$, $200_{1/2}$, $150_{2/3}$, $400_{3/1}$, $20_{3/4}$, $70_{3/5}$, $95_{4/6}$, $135_{5/6}$, $40_{6/2}$, $50_{6/7}$}.

Assume that this graph represents the control-flow for a given application. In this scenario, each node represents a *basic block* of instructions of the code, and the edges represent the conditional/unconditional *branches* between basic clocks. Assume also that the frequency by which the branches are taken during execution is indicated by the *weight* associated with each edge of the graph. For instance, the *Adjacency Matrix* $A = (a_{ij})_{NxN}$ of G is defined by:

$a_{ij} := W_{i/j}$ $\Rightarrow$ if V_i and $V_j \in E$;
$\Rightarrow$ else “0”if;

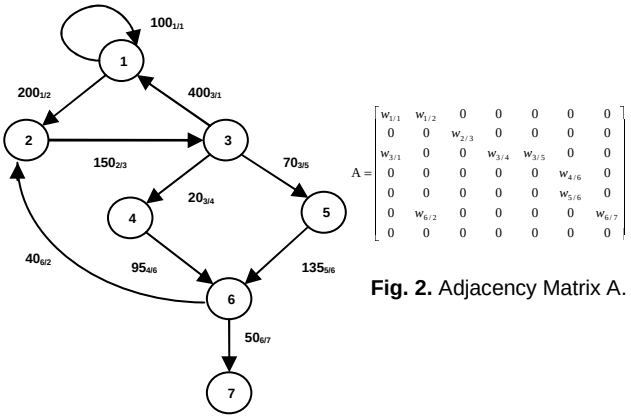
where N is the number of nodes in G. Then, Fig. 2 depicts A.

2.2. The Profiling Algorithm

Software profiling has been proposed by software engineers a few decades ago and has been used since then basically to observe, gather, and analyze data to characterize a program’s run-time behavior [3]. For these professionals, profiling deployed software is valuable because it can provide the meaning to improve programs after the development phase, when these programs are on the field. Then, profiling deployed software provides insights into how the software is actually utilized [4], which configurations are being employed [5], what development assumptions may not hold [6], where validation activities are lacking [7], or which scenarios are most likely to lead to a failure [8]. Profiling usually requires the instrumentation of the program, that is, the addition of probes to enable the examination of the program’s on-the-field run-time behavior. As such, the act of profiling penalizes the target

[1] *This work is partially supported by CNPq.*

software with execution overhead. The magnitude of the overhead depends, at least to some extent, on the number, the location, and the type of inserted profiling probes.



$$A = \begin{bmatrix} w_{1/1} & w_{1/2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & w_{2/3} & 0 & 0 & 0 & 0 \\ w_{3/1} & 0 & 0 & w_{3/4} & w_{3/5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{4/6} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{5/6} & 0 \\ 0 & w_{6/2} & 0 & 0 & 0 & 0 & w_{6/7} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 2. Adjacency Matrix A.

Fig. 1. Example of a graph terminology.

In the present work, we propose to use *profiling deployed software* in a quite different manner and purpose: in our case, this technique is used to provide insight into which parts (paths, functions, etc) of the program are being the most exercised. In other words, we assume that the scenarios that are most likely to lead to a system failure are those where the processor spends most of the execution time. Once identified, these parts are the only to be protected by embedding software signatures of *Type (a)*, as described in Section 2. The remaining (less used) parts of the application program do not receive embedded signatures and maintain their original format. By doing so, we optimize the number of checkpoints to be inserted along with the application program during the software development phase. In practice, this solution minimizes the memory overhead required to store checkpoints, reference and control information while not sacrificing the fault detection capability provided by CFC approaches.

In our approach, the instrumentation of the program is very simple: it is based on the addition of counters at the points where the CFC signatures are embedded in the application program. In the software validation phase, these counters are used to determine the frequency by which the signatures are checked out during application runs, for a representative input data set. By the end of this process, after reading out all counters, the instrumentation is erased from the program. Therefore, there is no memory overhead or performance penalty induced by the used instrumentation.

Based on this approach, we expect to simplify the Adjacency Matrix by:

(a) Increasing the matrix "0"s population. This is obtained by reducing the number of edges connecting nodes in the graph. To reduce the number of edges, we first select arbitrarily a threshold weight, W_{th}, below which all weighted edges are eliminated from the Weighted Graph, so thus from the Adjacency Matrix. In this approach, W_{th} represents the frequency by which that edge of the graph was exercised during application runs, for a representative input data set. In other words, W_{th} is defined by the counter value embedded at that program basic block.

(b) Reducing matrix dimension from A_{NxN} to A_{MxM}, where M < N. This is obtained by reducing the number of nodes of the Weighted Graph. Nodes (i.e., program basic blocks) that do not have weighted edges leaving from or reaching it can be eliminated from the Weighted Graph, which implies the elimination of lines and columns in the Weighted Matrix.

Note that as long as nodes are eliminated from the Adjacency Matrix, program basic blocks are eliminated from the Weighted Graph. Also, if a weighted edge W_{ij} is changed by a "0" in the Adjacency Matrix, a conditional branch is eliminated from the Weighted Graph. In both cases, as consequence, embedded monitoring signatures are strategically removed from the application program.

3. FINAL CONCLUSIONS

We presented a new approach, Optimized Embedded Signature Monitoring – OESM, to perform on-line control-flow fault detection. The claimed approach's advantage is the ability to perform a profiling algorithm that analyses the control-flow graph of user programs in order to reduce the number of checkpoints (i.e., signatures) to be inserted along with the application code during the software development phase. For a given application, this reduction is ruled by the best trade-off between "*the minimum number of signatures*" for "*the maximum fault detection coverage*". The embedded signatures are checked at runtime by the processor against compilation-time pre-computed values every time the processor reaches these signature points.

In order to evaluate the validity of the proposed approach, experimental results have been carried out. The obtained results indicate that OESM minimizes the number of signatures embedded in the application code (thus, minimizing memory overhead and performance degradation) while maintaining the same (or at least a similar) fault-detection capability level when applied to conventional control-flow fault detection approaches. At present, a tool that partially automates the proposed approach is under development.

REFERENCES

[1] Bernardi, P.; Veiras Bolzani; L. M.; Rebaudengo, M.; Sonza Reorda, M.; Vargas, F. L.; Violante, M. A New Hybrid Fault Detection Technique for Systems-on-a-Chip. IEEE Transactions on Computers, Feb. 2006, Vol. 55, No. 2. pp.185-198.

[2] Bezerra, E. A.; Vargas, F.; Gough, M. P. Improving Reconfigurable Systems Reliability by Combining Periodical Test and Redundancy Techniques. Journal of Electronic Testing: Theory and Applications – JETTA. Kluwer Academic Publishers, New York, USA. Vol. 17, May 1st, 2001, pp. 163-174.

[3] Diep, M.; Elbaum, S.; Cohen, M. Profiling Deployed Software: Strategic Probe Placement. Technical Report CSE-05-08-01/CSE-2005-005, Dept. of Computer Science and Engineering, Univ. of Nebraska-Lincoln, Lincoln, NE, USA, Aug. 2005.

[4] Orso, A.; Apiwattanapong, T.; Harrold M. J. Leveraging Field Data for Impact Analysis and Regression Testing. Foundations of Software Engineering, ACM, Sept. 2003. pp. 128-137.

[5] Memon A.; Porter, A.; Yilmaz, C.; Nagarajan, A.; Schmidt, D.; Natarajan, B.; Skoll: Distributed Continuous Quality Assurance. International Conference on Software Engineering, May 2004. pp. 449-458.

[6] Elbaum, S.; Diep, M. Profiling Deployed Software: Assessing Strategies and Testing Opportunities. IEEE Transactions on Software Engineering, 31(4), 2005. pp. 312-327.

[7] Elbaum, S.; Hardojo, M. An Empirical Study of Profiling Strategies for Released Software and Their Impact on Testing Activities. International Symposium on Software Testing and Analysis, ACM, June 2004. pp. 65-75.

[8] Liblit, B.; Aiken, A.; Zheng, Z.; Jordan M. Bud Isolation Via Remote Program Sampling. International Conference on Programming Language Design and Impl., ACM, June 2003. pp. 141-154.