RECCO: a REliable c/c++ Code COmpiler for dependable applications

Alfredo BENSO, Silvia CHIUSANO, Paolo PRINETTO

Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso duca degli Abruzzi 24
I-10129, Torino, Italy
Email: {benso, chiusano, prinetto}@polito.it
http://www.testgroup.polito.it

Abstract[1]

The present paper proposes a C/C++ Source-to-Source Compiler able to increase the dependability properties of a given application. The adopted strategies are based on code re-ordering and variable duplication. The proposed approach can be applied to any C/C++ source code, and introduces code modifications that are transparent to the original program functionality. Moreover, it is portable to any platform. The RECCO tool, which fully automates the process, allows the user to trade-off between the level of dependability improvement and the performance degradation due to the code modification.

1. Introduction

The increasing complexity and quality required by the market is making the design of dependable digital systems a key problem from an economical point of view. Recently, it has been investigate the possibility of designing dependable systems using commercial hardware and software components off-the-shelf (COTS), not designed, individually, to guarantee the required dependability level. Designing hardware using COTS moves the dependability issue at the software layer, now in charge of also guaranteeing high levels of availability and maintainability. In this context, the *software fault tolerance* aims at addressing system failures caused by a *hard* or *soft error* appearing in the system hardware. For sake of space, in this short summary we do not report neither details nor references about the previous works in this area.

The present paper proposes a *pure software* and *fully automated* approach based on *code re-ordering* and *variable duplication* to detect run-time hardware errors. *RECCO* (*RE*liable *C*ode *CO*mpliler), an ad-hoc Source-to-Source C++ compiler, has been implemented in order to transforms any input C/C++ *source code* into an output C/C++ *reliable code*, properly modified to increase its dependability characteristics. The proposed approach proved to have a high detection capability of faults occurred in the program data segment. The source-code modifications are completely *transparent* to the target program functionality and can be applied to any target program. Moreover, the generated reliable C/C++ code is *portable*, i.e., it is independent from the native compiler used subsequently to generate the executable file. Both the time and memory overhead introduced by the approach are acceptable and, in any case, controllable by the user on the basis of a trade-off between fault coverage and performances. In particular, *RECCO* supports the designer in identifying both the most critical portions of *code* and the most critical *variables* and suggesting the best modification towards a reliable code. The proposed techniques have been evaluated on a complete set of four benchmarks using BOND, a proprietary Fault Injection tool running under Windows NT 4.0.

A detailed description of *RECCO* can be found in [1].

2. The RECCO tool

The RECCO Source-to-Source compilation process runs through the following three different phases.

In the first phase, named *Code Reliability Analysis phase*, RECCO evaluates the *dependability properties*

[1] This work was partially funded by the Italian Space Agency (ASI) under Research Grant #332, and by Politecnico di Torino (Italy) under the project Progetto Giovani Ricercatori 2000 "*Development of a software environment for highly dependable space applications*".

of the variables, sorting them according to their criticality for a correct program execution. For each variable a *reliability-weight* is computed, which takes into account the variable *lifetime* and its *functional dependencies* with other variables. The evaluation performed in this phase allows identifying the most *reliability-critical* variables and correctly driving the next two phases of the RECCO compiling process.

In the second phase, *Code Re-ordering phase*, RECCO modifies the original C/C++ code and generates a code, functionally equivalent to the original one, but significantly improved in terms of dependability characteristics. The adopted approach consists in performing local optimizations aiming at reducing the reliability-weight of the variables identified during the Code Reliability Analysis.

In the third phase, *Variable Duplication phase*, the dependability properties of the application source code are further improved by resorting to variable duplication. In the actual implementation of the tool, the user can specify the percentage of variables to be duplicated, and RECCO selects, among all the variables, the ones that are more critical for the code security.

3. Experimental results

RECCO has been developed resorting to the C/C++ parser, and a C/C++ lexical analyzer by James Roskind [2]. To assess the effectiveness of the proposed techniques in improving the dependability properties of the code, benchmark programs have been executed inside BOND, a proprietary fault injection environment.

The considered test cases comprehend a *matrix product*, an *elliptical filter*, and two ad-hoc developed examples, *Ex#1* and *Ex#2*, executing a sequence of floating point operations. Concurrently with the program execution, the Fault Injector emulates the appearance of data faults corrupting memory locations storing data. The adopted fault model is the transient fault *Single Event Upset* (SEU), consisting in temporally flipping one bit in one data memory location. Each experiment involved the injection of a *fault list* including 1,000 faults, each characterized by an *injection location* (i.e., a bit position inside the memory array), and an *injection time* expressed as the elapsed time from the beginning of the program execution. Both the parameters have been randomly selected. The system reaction to the injected faults has been classified in one of the following categories: *Fail Silent* (FS), *Fail-Silent Violation* (FSV), or *Detected* thanks to the introduced code modifications.

Results were obtained executing RECCO on a 333Mhz Pentium II with 64Megabytes of RAM and running Windows NT 4.0.

The Fault Injection experiments results collected in Table 1 show that, using code re-ordering, the occurrences of FSVs decreases: part of the faults causing FSVs in the original program is moved to the FS condition. The FSVs is reduced about 5% for the *elliptic filter* and 9% for the *Ex#2*. On the *Ex#1*, where the few dependencies existing among the variables allow RECCO to fully re-scheduling the execution flow, the FSVs reduction reach the 65%. On the *matrix product*, instead, the code re-ordering is not applicable, since the existing high dependencies do not allow code relocation.

The FSVs further decrease when the code re-ordering technique is coupled with the variable duplication one: the reduction of FSVs ranges from 96% to 100%. According to the percentage of duplicated variables, a subset of the FSVs is intercepted and the occurrence is notified to the external application. Results showed on Table 1 are obtained duplicating all the variables. The number of FSVs could however be still different from zero if a fault affects a variable immediately after its correctness has been verified, but before it is read by other variables (for example in the case of *matrix product* and *Ex#1*).

4. References

[1] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri, *A C/C++ Source-to-Source Compiler for Dependable Applications*, IEEE International Conference on Dependable Systems and Networks (FTCS 30), New York (NY), June 2000

[2] ftp://ftp.sra.co.jp/.a/pub/cmd/c++grammar2.0.tar.gz

Table 1: Improvement of dependability properties using RECCO

	Original program		Using code re-ordering			Using code re-ordering & all variables duplications			
Benchmark	**FS**	**FSV**	**FS**	**FSV**	**Reduction**	**FS**	**Detected**	**FSV**	**Reduction**
Matrix product	751	249	751	249	0 %	999	248	1	99.59 %
Elliptic Filter	967	43	969	41	4.65 %	969	41	0	100 %
Ex#1	795	205	815	185	9.75 %	815	178	7	96.58 %
Ex#2	838	162	944	56	65.43 %	944	56	0	100 %