ETFA'2009

2009 IEEE Conference on

Emerging Technologies & Factory Automation

September 22-26, 2009 @ Mallorca, Spain

Welcome Messages

Keynotes

Author Index

Committees

Program at a Glance

Sponsors

Local Information

Technical Program

Reviewers List

© 2009 IEEE IEEE Catalog Number: CFP09ETF-CDR ISBN: 978-1-4244-2728-4 ISSN: 1946-0759
Edited by Antoni Grau, Javier Campos and Gabriel Oliver

IEEE Celebrating 125 Years of Engineering the Future

Universitat de les Illes Balears

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Automated Software Diversity for Hardware Fault Detection

Gerald Gaiswinkler
Elektrobit Austria GmbH
Kaiserstrasse 45 / Stiege 2
1070 Vienna, Austria
gerald.gaiswinkler@elektrobit.com

Andreas Gerstinger
Institute of Computer Technology
Vienna University of Technology
Gusshausstrasse 27-29
1040 Vienna, Austria
gerstinger@ict.tuwien.ac.at

Abstract

Software in dependable systems must be able to tolerate or detect faults in the underlying infrastructure, such as the hardware. This paper presents a cost efficient automated method how register faults in the microprocessor can be detected during execution. This is done with the help of using compiler options to generate diverse binaries. The efficacy of this approach has been analyzed with the help of a CPU emulator, which was modified exactly for this purpose. The promising results show, that by using this approach, it is possible to automatically detect the vast majority of the injected register faults. In our simulations, two diverse versions have – despite of experiencing the same fault during execution – never delivered the same incorrect result, so we could detect all injected faults.

1. Introduction

If a computer system is used in a availability or even safety-critical application, mechanisms must be implemented to prevent catastrophic failures. This paper introduces a software based mechanism – automated software diversity - to aid in the detection of hardware faults. The main focus is the detection of faults and not necessarily to tolerate the hardware fault. In case such a fault is detected, an appropriated reaction has to be initialized, which can be the continuation of system operation in a degraded mode, the activation of a backup system, or in some safety-critical applications, the system can be transferred into a fail-safe state. In the latter case, safety of the safety-critical system is maintained, but availability is lost. The idea of the automated software diversity as it is used in this paper is depicted in figure 1.

As shown, the compiler is used to generate different variants based on the same program: The compiler provides different options to influence the compilation process. Thus the compiler can be used to compile a program repeatedly with different options to generate two or more variants of the same program. Hence, functionally equivalent but structurally different variants of a program are created. The program variants use hardware components during execution of the program in a different way. Because a hardware fault affects the variants differently, the fault can be detected by comparing the results of the program execution.

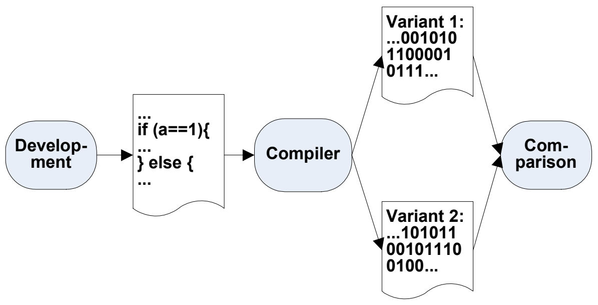

Figure 1. Automated software diversity

The paper is structured in the following way: Section 2 describes related work. Section 3 presents the method and the test environment we have used for our experiments. Sections 4 and 5 describe the results: the static comparison of the binaries, and the dynamic simulation results, respectively. Finally, section 6 gives a summary and a conclusion.

2. Related work

Diversity as a method to tolerate or detect faults has been in use since the 1970s. Mostly, it is used on a design level, in order to be able to tolerate (software) design faults. N-version programming (the generation of N functionally equivalent software versions by N non-communicating teams) is a typical application of diversity. A good survey of the various forms of diversity is contained in [1]. Typical applications of diversity can be found in the aircraft [2] and railway domain [3].

Various software methods to detect hardware faults are described in [4]. The specific idea of using automated diversity for hardware fault detection was explored in [5] and [6]. However, in these cases, explicit "diversification tools" have been implemented. Although the results were promising, the portability of this solution is severely limited if dedicated tools have to be used.

The purpose of this paper is the investigation, if with the help of diversity automatically generated by compiler options, a similarly promising result can be obtained. This was motivated by the increasing complexity of compilers, which offer a large number of configuration options, which significantly influence the generated machine code.

3. Method

This section introduces the basics of the compiler, used to generate the variants of the test programs. Then, we describe the faults that we anticipate to be able to detect, and the test programs and test environment used to validate our anticipation.

3.1. Compiler

The whole compilation process can be divided into four successive stages: pre-processing, compilation, assembly, and linking [7]. Each stage provides options to influence the behaviour of the respective stage. A part of the available options at the compilation stage are summarized as options for optimization of the compiler output. These optimize options can be used to improve the performance and/or the code size of the given program; in our case, these optimize options are also used as a method to automatically generate different variants of a given program. During this work, two different compilers were used: The first compiler is the GNU compiler collection gcc (version 4.2.2) [8]. This compiler provides five different levels of optimization. The first level (level 0) does not perform any optimization and the last optimization level (level 3) activates 48 optimization flags to improve the performance and the code size of the generated executable.

The usage of an alternative compiler is another way to produce a different variant of the same program. To test the automated software diversity with two different compilers, the Intel compiler collection icc (version 10.1) [9] was used to generate an another variant.

3.2. Considered faults

During the following analysis, only a subset of the vast number of possible microprocessor faults can be considered. Our fault hypothesis mainly includes various bit faults in the general purpose registers of the microprocessor. Transient bit faults as well as permanent "stuck-at" bit faults are simulated and analyzed. A transient bit fault changes the value of a bit in a register, but does not permanently damage a register. A permanent stuck-at bit fault leads to the fact that a certain bit is permanently set to 0 or 1 ("stuck-at-x").

As described in [5] and [6], transient hardware faults can be detected by repeated execution of the test program, because a transient fault corrupts only one execution; a write operation at the faulty position removes the faulty state of the position. By comparing the results of the program executions, which may be different due the fault, the fault can be detected. In contrast to transient faults, a permanent fault cannot be removed by a write operation and affects the two different variants in the same way. Thus, the main focus lies in the analysis of detection of permanent stuck-at faults. The appearance of a fault (when and where it appears) is specified with the help of probabilities. This probability is configured, that at least one bit fault happens during program execution.

3.3. Test programs

Two different test programs are used to analyze the effect of the injected hardware faults during the execution of different program variants. Additionally, it is to be examined whether programs with a special structure are better suitable for the automated software diversity.

For this reason, two test programs were chosen, which are algorithmically very different. The first test program implements the quick sort algorithm. This is done in a recursive way. The program consists of a high number of jump instructions, many function calls, but does not contain many calculation blocks.

The second test program implements the data encryption standard (DES) to cipher and decipher 8 random characters. To implement DES, large calculation blocks are necessary but fewer function calls and jumps.

3.4. Test environment

In order to analyze the effect of a hardware fault, we use the method to inject these faults during execution into the different variants of the test programs and record the consequences. As true hardware fault injection is impractical, we use software simulated hardware fault injection. To produce these simulated hardware faults, we have extended the machine emulator QEMU (version 0.9.0) [10] with the possibility of fault injection[1]. With the native QEMU, it is possible to run a program for a specific microprocessor (host PC e.g. SPARC) on a different microprocessor (target PC e.g. x86). Each instruction from the host PC is translated into an instruction of the target PC. This translation step is used to inject artificial hardware faults into the translated code of the target PC. QEMU was extended for the x86 microprocessor: Currently, only if the target CPU and the host CPU are configured as x86 the hardware fault injection mechanism can be used.

With our extended version of QEMU, it is possible to inject bit faults in all general purpose registers and into the status flags (the EFLAGS register). Furthermore, it is possible to manipulate the first 2 bytes of a fetched instruction, in order to simulate the execution of a wrong instruction. This mechanism is currently only implemented for a small subset of the x86 instruction set.

[1] Available at: http://www.ict.tuwien.ac.at/sysari/FEMU

The duration of the bit fault can be configured to be permanent or transient. Our extended QEMU allows to configure the appearance of a fault by specifying a fault probability. This mechanism is implemented by using a random number generator. By using the same fault probability and the same seed for the random number generator, it is possible to inject the exactly same fault into different variants of the test program.

With these extensions of the machine emulator QEMU, it is possible to simulate the considered faults as denoted in 3.2.

The QEMU test environment is integrated into the test framework as depicted in figure 2.

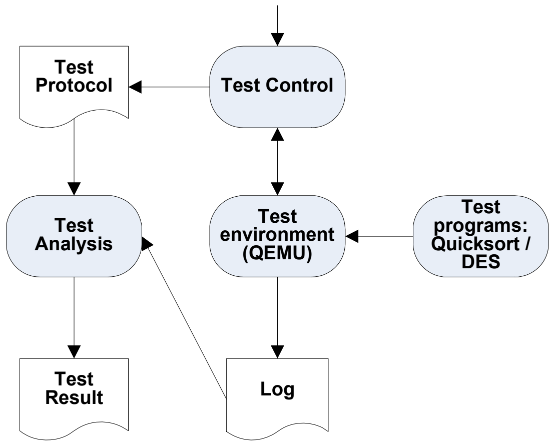

Figure 2. Test framework

The test framework is used to inject the same hardware fault into two different variants of a test program and record the consequences. The test control script coordinates the execution of the test environment i.e. it configures the fault probability and the seed and executes the variants of the test program. Furthermore, the test control script creates a test protocol.

The test protocol, together with the log files, is used by the test analysis script to generate the test result. These test results are the basis for the dynamic analysis.

4. Static analysis

Before performing the dynamic analysis, we have compared the generated variants statically. This has included the analysis of the number of executed instructions and the usage of the general purpose registers. The insight gained from this static analysis helps in understanding different behaviour in response to faults. As described in section 3.1, the compiler gcc was used to generate four different variants for each test program. For this purpose, the optimization levels 0 to 3 were applied to generate the four variants. Another variant was generated by the usage of the compiler icc with the default optimization level. These five variants were used for the following analysis.

4.1. Quick sort

The first considered parameter is the number of executed instructions. The corresponding number for each variant is depicted in table 1.

Compiler	Optimization	Quick sort	DES
gcc	Level 0	1 151 252	2 153 018
gcc	Level 1	1 082 859	1 256 772
gcc	Level 2	1 074 244	1 238 185
gcc	Level 3	1 069 738	917 098
icc	Default	982 973	815 218

Table 1. Number of executed instructions

The number of the executed instructions includes also the instructions which are executed inside e.g. system calls, and the repeated execution of instructions inside loops. Therefore, this number represents the real number of executed instructions and not only a analysis of the static code.

As shown in table 1, the number of executed instructions for both programs can be reduced by using higher optimization levels. However, the reduction for the quicksort program is much smaller. The smaller reduction can be explained with the recursive structure of the program. It is not possible for the compiler to simplify function calls and loops even more. This behaviour can also be observed during analysis of the usage of the general purpose registers. The variants rather equal often access – read and write access – the registers, except the registers EAX and EBP. These two registers are depicted in Figure 3.

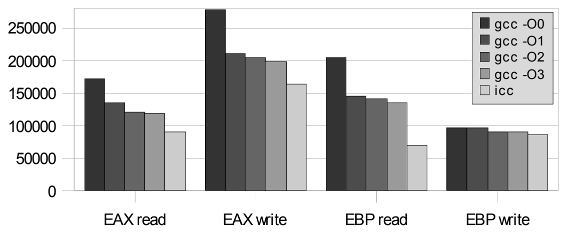

Figure 3. Register access (quick sort)

The biggest differences are observable at the read and write access of the EAX register, between the variant compiled with gcc without optimization and the variant compiled with the compiler icc. There are also measurable differences at the read access of the EBP register which is used by function calls. All other observed values of read or write access are almost equally distributed between the variants.

In contrast to the first test program, the second test program – data encryption standard – showed a different behaviour during code comparison.

4.2. Data encryption standard

As described in section 3.3, this test program consists of many calculation blocks. The optimization mechanisms of the compiler can simplify these blocks in order to optimize the performance of the program. The effect of the optimization can be shown by comparing the number of executed instructions of each variant.

Table 1 shows the number of executed instructions for each variant of the second test program (DES). The optimization levels had a huge impact on the number of executed instructions. Due to the usage of optimization (level > 0), the number of instructions can be nearly cut by half.

The compiler icc with the default optimization level leads to the "best" result. As described in the documentation of the compiler icc [9], the default optimization level can be compared with the optimization level 2 of the compiler gcc.

Due to the huge differences between the number of instructions of the variants, also differences in the usage of the registers can be observed.

The variant without optimization performs read operations on the registers EAX, ECX and EDX most frequently, but uses some other registers e.g. EBX rarely.

The usage of optimization distributes the read operations more on the registers. This characteristic can also be observed at the write operations. Furthermore the usage of optimization leads, in contrast to the first test program, to more differences in the usage of the registers between the variants. This fact is depicted in figure 4.

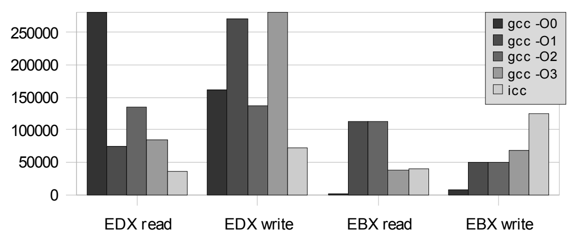

Figure 4. Register access (DES)

The two registers, as shown in figure 4, are a good example to show the differences between the variants. In contrast to the first test program, the usage of all registers shows a significant difference.

In summary, the optimization options have a bigger impact on the second test program. Due to the structure of this program, the compiler can create more diverse variants. This was shown by comparing the variants regarding the number of executed instructions and the usage of the general purpose registers. The biggest analogy between two diverse variants was between the variant compiled with gcc optimization level 1 and the variant compiled with gcc optimization level 2. During the dynamic analysis it will be shown that these diverse variants still lead to a different reaction by injecting the same hardware fault.

5. Dynamic analysis

As described in section 3.4, the developed test environment was used to inject simulated hardware faults into the diverse variants. The first part (see section 5.1) of the dynamic analysis covers random transient bit faults. The second part (see section 5.2) analyses random permanent hardware faults. It is important, that the same random permanent fault is injected into the two considered diverse variants. This is necessary to analyze the efficacy of the automated software diversity with respect to detecting permanent hardware faults.

During each execution of a variant, at least one transient or permanent bit fault was injected into the general purpose registers and the reaction of the variant was recorded. In order to test only the diverse algorithm, the fault injection starts at the first instruction of the main function of the variant and stops before the output of the results.

5.1. Transient register bit faults

During the analysis, the following parameters were recorded and examined: The first parameter is the state of the program after execution. This state can be *safe, crash, false* and *loop*. The state safe means, that the execution reached the end of the program and outputs the correct result. If the program crashes because of an injected fault, the program enters the state crash. The worst case is the state false: The execution reached the end of the program but outputs an undetectably incorrect result. If a fault influences e.g. a count register of a loop, the execution can get stuck in an endless loop. The test environment interrupts the execution after 4 000 000 instructions and sets the state of the program to loop.

The second parameter is the latency. This parameter represents the number of executed instructions between the injected fault and the reached program end state crash or false.

As an example of this analysis, the results are depicted in table 2 and table 3 for the variants compiled with gcc optimization level 0 and the variants compiled with icc for both test programs. For each variant of the test programs 1000 test runs were performed, but only runs with at least one fault injection were considered.

State	Quick sort gcc -O0	Quick sort icc	DES gcc -O0	DES icc
safe	76.3%	73.2%	83.7%	79.4%
crash	22.9%	24.1%	13.1%	10.1%
false	0.6%	2.5%	2.9%	8.4%
loop	0.2%	0.2%	0.3%	2.1%

Table 2. Reached end states

As shown in table 2, the injected transient bit fault has no effect on the majority of the test runs. The fault can be tolerated by the execution and therefore the execution reaches the end of the program. If the fault has an effect on the execution, the variants of the test program quick sort crash more often than the variants of the test program DES. It is also interesting to note that the variant compiled with icc of the test program DES is more vulnerable to produce a false result than the variant compiled with gcc.

This effect is caused by the structure of the test programs: The destination addresses of jump instructions are stored in the registers. The test program quick sort uses various jump instructions, thus a corrupted register leads to a jump to an invalid address, so that the program crashes more easily.

The latency of an injected fault is the time between the fault injection and the effect of the fault (i.e. the crash event or the production of a false result). This latency for the various test programs is depicted in table 3.

Latency (instructions)	Quick sort gcc -O0	Quick sort icc	DES gcc -O0	DES icc
1-5	84.4%	80.2%	92.3%	91.2%
6-15	3.5%	4.6%	2.8%	1%
16-40	3.8%	4.6%	2.9%	3.9%
41-100	2.1%	3.9%	1.6%	2.7%
> 100	5.8%	6.7%	0.4%	1.2%

Table 3. Latency

It is noteworthy that in a large majority of the cases (~82% by quick sort and ~91% by DES) the program terminates practically immediately: only 1 to 5 instructions are executed after the fault has been injected.

The next analysis deals with the method of automated diversity to detect hardware faults.

5.2. Random permanent register stuck-at faults

During the analysis, the diverse variants of the test programs were used to investigate the potential of the automated software diversity for hardware fault detection. As described in section 3.4, the test framework was used to inject the same hardware fault into two diverse variants and record the results. Identical permanent stuck-at faults (identical meaning at the same point in time and corrupting the same bit in the register) were injected into both variants.

The test framework, with two variants of the test program, was used to simulate the idea of the automated software diversity as depicted in figure 1.

By comparing the results of the variants, differences can be detected. In a real safety-critical system, the detection of a difference in the results can lead to a transition to a fail-safe state. This means that even if both variants produce incorrect results, one can still achieve a safe state as long as the (incorrect) results are different.

During the test, the final states (see section 5.1) of the considered variants were recorded and compared. If the injected fault has lead to the state false in both variants, the results of the variants were also compared in order to detect the worst case scenario: If both variants deliver the *same* false result, the hardware fault cannot be detected and the program uses the wrong values for further operations.

For each test program, the combination of the variants gcc -O0 versus gcc -O1, gcc -O0 versus gcc -O3 and gcc -O2 versus icc were considered. The probability for a fault injection was set in order to inject at least one stuck-at fault during execution. Because of the different number of executed instructions in the two variants, it is possible that no fault is injected into the second variant.

For each combination we have performed at least 750 test runs.

5.2.1. Quick sort

The first considered test program is the recursive quick sort algorithm. As an example of this test program, the results of the combination of the variants gcc –O0 and gcc –O3 are depicted in table 4.

State	Quick sort gcc -O0	Quick sort gcc –O3	Quick sort both
crash	89.5%	87.4%	82.7%
false	0.0%	0.3%	0.0%
loop	1.1%	1.5%	0.3%
safe	9.4%	7.1% (3.7%[2])	5.0%

Table 4. gcc –O0 vs. gcc –O3 (Quicksort)

For this combination we used the variant without optimizations (gcc –O0) and the variant with the highest optimization level (gcc –O3) of the compiler gcc. The last column in table 4 shows the number of test runs where both variants entered the same final state.

As shown in table 4, the injected permanent fault leads in most cases to a crash of both variants. Rarely one variant outputs an incorrect result. By comparison with the result of the second variant, the false result was detected. The high number of crashes is caused by jumping to invalid destination addresses (see section 5.1).

By comparing the other combinations, we identified the same behavior as depicted in table 4. In the majority of the test runs (~83%), both variants crashed.

Due to the structure of the test program (see section 4.1), the compiler has fewer possibilities to generate

[2] In these cases, no fault was injected due to the difference in the number of instructions (O3 generates fewer instructions than O0) and so the variant entered the state safe.

diverse variants. Therefore the variants show the same behavior after injecting of a permanent stuck-at fault.

5.2.2. Data encryption standard

As described in section 4, the compiler can generate "more" diverse variants of this test program than for the first test program. This fact has a high impact on the behaviour of the variants in case of permanent faults. Also the distinct structure of this test program leads to different results between the variants.

In table 5, the results of the combination of the variants gcc –O0 and gcc –O3 for DES are shown.

State	DES gcc –O0	DES gcc –O3	DES both
crash	62.1%	34.7%	27.1%
false	15.1%	7.5%	3.1%
loop	5.3%	8.1%	1.4%
safe	17.5%	6.2% (43.5%)	2.6%

Table 5. gcc –O0 vs. gcc -O3 (DES)

Similar figures are observed when using different compilation options or even compilers. In table 6, the results of the combination of the variants gcc –O2 and icc are shown.

State	DES gcc –O2	DES icc	DES Both
crash	69.2%	47.1%	39.5%
false	12.0%	8.3%	3.5%
loop	7.2%	9.2%	1.6%
safe	11.6%	6.8% (28.6%)	3.1%

Table 6. gcc –O2 vs. icc (DES)

In comparison with the test program quick sort, the number of incorrect results is considerably higher. This behaviour is caused by the structure of the program. A fault in a register rather leads to faulty calculation than to a jump to a wrong location. The different numbers of executed instructions also affect the result. Due to the fact that the probability of a fault injection per instruction is constant, in a variant with fewer executed instructions the probability of fault injection within the execution time is smaller.

In the case, where both variants provide an incorrect result, these results are compared to detect the worst case scenario (same incorrect results). This worst case scenario never occurred. Therefore, the method of automated software diversity allowed us to detect all injected hardware faults.

6. Conclusion

With the help of the static code comparison we have shown that the compiler has the potential to generate diverse variants of the same program by using the options of the compiler. The degree of diversity depends highly on the structure of the program. A program with frequent calculation blocks is more suitable to generate diverse variants with the method of automated software diversity. To prove the hardware fault detection function of this method, we extended a microprocessor emulator with the option of fault injection into registers. The advantage of this approach is the high flexibility and repeatability of the experiments. The only disadvantage of this approach is the relatively high duration of the test runs. This was due to the fact that some optimizations of the emulator were disabled to facilitate the fault injection mechanism, causing an increase in the emulation time.

Furthermore, we could show that all artificial injected hardware faults were detected due to the automated software diversity. The same injected permanent fault influenced the two variants differently in all cases. This means that by comparing the results of the test programs, all injected faults were detected. During this work, we have considered only register faults. The extended emulator (and other existing methods) can be used to investigate other faults.

Also the provided options of the compilers can be the subject of further research in order to generate diverse programs.

In summary, the method of automated software diversity is a fast and cost efficient method to increase the dependability of critical systems. However, the specific algorithm of the application should be taken into account, in order to ensure its effectiveness.

References

[1] Bev Littlewood, Lorenzo Strigini. A discussion of practices for enhancing diversity in software designs. DISPO project technical report, Centre for Software Reliability, City University, 2000.

[2] Y. C. (Bob) Yeh. Design Considerations in the Boeing 777 Fly-By-Wire Computers. The 3rd IEEE International Symposium on High-Assurance Systems Engineering. 1998.

[3] Heinz Kantz, Christian Koza. The ELEKTRA Railway Signalling System: Experience with an Actively Replicated System with Diversity. Proceedings of the 25th International Symposium on Fault-Tolerant Computing. 1995.

[4] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, Massimo Violante. Software-Implemented Hardware Fault Tolerance. Springer, Berlin. August 2006.

[5] Tomislav Lovric. Systematic and Design Diversity – Software Techniques for Hardware Fault Detection.

Lecture Notes In Computer Science, LNCS Vol. 852, Springer Verlag. 1994.

[6] Markus Jochim. Detecting Processor Hardware Faults by Means of Automatically Generated Virtual Duplex Systems. Proceedings of the International Conference on Dependable Systems and Networks (DSN'02). IEEE, 2002.

[7] Brian Gough, Richard Stallman. An Introduction to GCC. Network Theory Ltd.. March 2004.

[8] Free Software Foundation Inc. GCC, the GNU Compiler Collection, May 2009. http://gcc.gnu.org/.

[9] Intel Corporation. Intel C++ Compiler 10.1, May 2009. http://www.intel.com/cd/software/products/asmo-na/eng/compilers/clin/277618.htm.

[10] Fabrice Bellard. QEMU, May 2009. http://bellard.org/qemu/index.html.