A Detector for Harmful Errors

Jing Yu and María Jesús Garzarán[†]

Google Inc.
jingyu@google.com

[†] University of Illinois at Urbana-Champaign
garzaran@illinois.edu

Abstract

This paper describes the strategies that we follow to design a system for the detection of faults that are likely to cause Silent Data Corruptions (SDCs). To design such a system we follow an empirical approach by which values computed by the program are randomly perturbed and the effect of these perturbations on the correctness of the program output is observed. This approach is based on the observation that only a small fraction of faults result in SDCs, indicating that SDCs are caused by the corruption of a small fraction of variables. To avoid unnecessary work, our system only processes those faults that produce incorrect results and ignore those faults that, although changing the intermediate results, do not affect the final outcome of the program. Experimental results with `gzip` *and* `twolf` *show that by monitoring the values of a few variables our system can detect more than 40% of the SDC errors, while inducing negligible overhead.*

1. Introduction and Motivation

As semiconductor technology scales into the deep submicron regime the occurrence of transient errors will become more frequent. This situation demands new error detection strategies that require little hardware modification and have low performance overhead. To design such strategies, we should make use of the fact that only a small fraction of the faults result in program errors. As resported in [11] we have observed that only 6% of the injected faults result in Silent Data Corruptions (SDC). Although with differences, other groups have obtained similar results, e.g. [8]. SDC are those errors where the program finishes normally but the produced output is incorrect. SDC is the most serious type of errors, because the user gets an incorrect output and does not know about it. Our experiments also showed that about 19% of the errors resulted in segmentation faults, while 3% were detected by assertions inserted by the programmer. These last two types of errors are less harmful because the abnormal end of the program indicates a problem. By re-executing the code that ended abnormally from the last checkpoint it is possible to determine what caused the problem: a bug if the same error appears again or a soft error when it does not[1].

A large fraction of the faults we injected (72% in our experiments) did not manifest in any way. The errors generated by these faults are called unACE (the bit was unnecessary for Architectural Correct Execution [4]). There are several possible reasons for this. One is that the fault was injected into a register whose content was not used after the injection (dynamically dead) or that the fault did not change the result of the operation applied to that variable (for instance a fault injected into a variable used in the comparison of an `if` condition will not change the program output if the error does not modify the outcome of the `if` condition). Also, sometimes the program takes the wrong path, but later converges to the correct path, and the additional instructions executed do not affect the program output.

Previous error detection approaches did not take advantage of the very low frequency of SDC errors and the high frequency of unACE errors. For this reason, fault-tolerant systems developed following these earlier strategies replicated execution, either in hardware or in software, and inserted checks before certain synchronization instructions [11, 7, 5]. Although these systems could detect all the SDC errors (and a large fraction of the unACE errors), they have a large performance overhead and/or hardware cost. Similarly, the approaches that detect likely invariants or symptoms [9] are designed to detect invariants without knowing whether they detect the harmful SDC or the innocuous unACE errors.

The goal of the system outlined in this paper is the detection of only those faults that cause SDC errors. The paper is organized as follows. Section 2 presents an overview of the system, Section 3 shows our results, Section 4 presents related work, Section 5 presents future research directions and Section 6 concludes.

[1] When a program has a heisen bug, it is possible that the error does not occur again when re-executing the code.

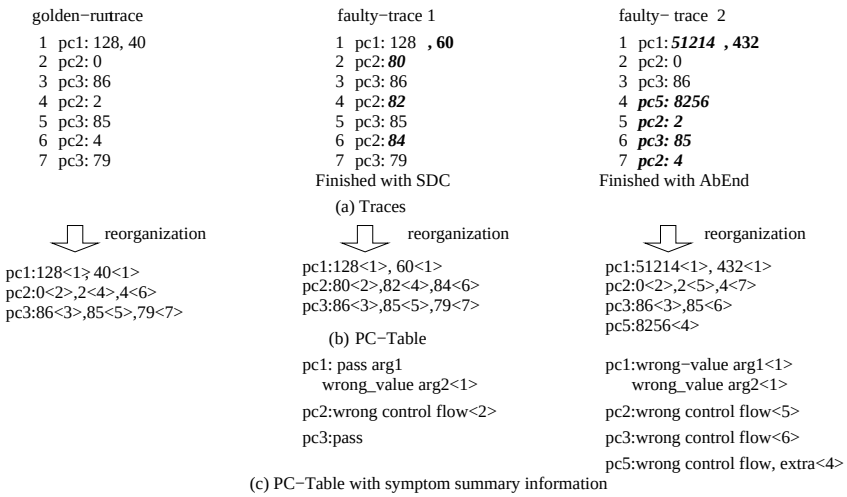

Figure 1. Example of trace reorganization and comparison.

2. Overview of the System

The goal of our approach is to detect those errors that are likely to cause SDCs. We make use of the observation that for an error to manifest, it must eventually corrupt the output of a program instruction. Thus, our intuition is that there must be a set of variables that when corrupted are more likely to cause SDCs, while a different set is more likely to cause segmentation faults, and yet another set is more likely to cause unACE errors. We do not detect errors in the last two categories. As we said above, unACE errors do not need to be detected and segmentation errors will warn the programmer who can then re-execute the code from the last checkpoint. Consider for example an error in a variable that is used to address memory. This error is likely to cause a segmentation fault, specially when the error affects the most significant bits; Therefore, since it does not lead to a SDC error, we do not attempt to detect when this type of variables are corrupted. Once we know what are the critical variables, we should be able to find efficient mechanisms to detect when these variables are corrupted.

We have built a system that detects the *SDC-indicating instructions*. To this end we follow an empirical approach that injects a single fault into a random location in the binary program and let the program run until it completes. The system tracks how the injected error propagates during program execution by collecting traces. The traces record the output of the instructions whose results are potentially visible outside the program. These instructions are all the stores and the system calls that generate an output to a file or to the screen, such as `printf`. For the stores, we record the data value and the data memory address, while for the system calls we record the arguments. At the end of the run, we compare the program output with the correct output (the one resulting from an execution without faults or golden-run) and classify the run: 1) unACE (program outputs were identical), 2) SDC (program outputs are different, but the program finished without any visible symptom), 3) abnormal termination. This last category includes the cases of segmentation fault, failed program assertions, infinite loop, or any other abnormal program exit. Our system repeats this experiment thousands of times. The system then identifies the store instruction operands or the system call arguments that were found more frequently corrupted in the traces associated with SDC errors.

Next, we give a detailed description of how the system works. In Section 2.1 we explain the symptoms we extract from the traces, while in Section 2.2 we explain the metrics that we compute to determine which are the SDC-indicating instructions.

2.1 Symptoms extracted from traces

As explained above, for each experiment we collect a trace. Traces contain information only of those instructions that produce a visible output. Thus, we collect data value and data memory address for stores and system call arguments for those system calls that produce an output to the screen or to a file. In addition, each entry in the trace includes the Program Counter (PC) and the serial number (SN). SN indicates the location of the entry in the trace. Traces do not collect information about register spills. For stores of heap allocated variables, only the data value is recorded, as the addresses of heap allocated variables may change from run to run. We also classify the run as SDC, unACE or abnormal termination. An example is shown in Figure 1-(a).

After a faulty trace has been collected, we need to compare it with that of the golden-run. For the comparison,

	wrong value arg1			wrong value arg2			wrong control flow		
PC	AbEnd	SDC	unACE	AbEnd	SDC	unACE	AbEnd	SDC	unACE
PC_1	1	0	0	1	1	0	0	0	0
PC_2	0	1	0	-	-	-	1	0	0
PC_3	0	0	0	-	-	-	1	0	0
PC_5	0	0	0	-	-	-	1	0	0

Table 1. Symptom Table

we reorganize the information stored in the trace. We build a PC-Table with entries for all the PCs that appear in the trace. Each PC has an entry with a list of tuples. The tuples repeat the information stored in the table for that PC. Each tuple contains the value manipulated by the instruction at that PC plus the SN of the instruction in the trace. An example of this process is shown in Figure 1-(b). A PC-Table is also built for the golden-run trace.

Next, we compare each entry in the PC-Table of a faulty-run with the entries of the golden-run. For each PC in the PC-Table we record one of the following symptoms:

• "pass": when the value and serial number are identical in both tables.

• "wrong value", when there is a mismatched value.

• "wrong control flow", when serial numbers are different or when the golden-run PC-table has an entry that is not found in the faulty one or viceversa.

When wrong value or wrong control flow is recorded, we save the SN of the first entry with the mistmatched value or mismatched SN, respectively. When a PC has both the wrong value and the wrong control flow symptoms, we only record the symptom that appears first. Notice that the PC-Table contains a tuple for each data value recorded for each instruction. When we compare the faulty-run with the golden-run, we record pass or wrong value for each value recorded for that instruction, e.g., data and address for stores. At the end of this stage we have the PC-Table with the symptom summary information (pass, wrong value or wrong control flow), as shown Figure in 1-(c). In addition, we know if the run finished abnormally, with SDC or the error was unACE. Next, we combine this information together in the Symptom Table shown in Table 1. The Symptom Table has an entry for each PC in the program. Each PC entry contains an entry for wrong control flow plus as many entries for wrong value as operands or arguments the instruction with that PC has. Each of these symptoms contains a counter for abnormal termination, SDC and unACE.

2.2 Metrics

The Symptom Table contains the information that we need to determine which are the SDC-indicating variables. It also contains what symptom is the best to predict the SDC error, a wrong value in a particular operand or argument of the instruction or a wrong control flow. To find the SDC-indicating variable, for each pair PC and symptom we compute the following metrics:

- $SDCcoverage$, which measures the fraction of SDC-errors that can be detected by the given instruction and symptom. It is computed as $SDCcoverage_{PC,symptom_i} = SDC_{PC,symptom_i}/totalSDC$, where $SDC_{PC,symptom_i}$ is the number of SDC errors observed in the instruction with program counter PC and symptom *i*.

- *SDCdistinguishability*, which measures how well this SDC-indicating instruction can distinguish SDC-errors from other types of errors $SDCdistinguishability_{PC,symptom_i} = SDC_{PC,symptom_i}/(SDC_{PC,symptom_i} + unACE_{PC,symptom_i} + AbEnd_{PC,symptom_i})$, where the denominator is the total number of errors observed by the instruction with program counter PC and symptom *i*.

- *Protection cost* measures the cost to protect a variable.

Our goal is to find a small set of SDC-indicating instructions with high SDC-coverage, high SDC-distinguishability, and low protection cost. In our experiments we first selected those variables with high SDC-distinguishability. For each pair instruction and symptom we need to find an inexpensive error detection mechanism. When the symptom is "wrong value", we profile the code to determine how easy is to find a likely-invariant checker [9] or a perturbation-based checker [6] or another kind of value predictor for that instruction. If this does not work, then we try to replicate the instructions in the backward slice for this variable. When the symptom is due to wrong control flow, we need to trace back from the chosen instruction to find the place that makes the control flow to take the wrong path. Our traces only collect store operands and system call arguments, but an error in control

flow can be due to an error in a variable that is never stored. Thus, to protect the chosen instruction we may need to detect and protect critical variables that are only loaded, and never stored.

Notice that several instructions can detect the same SDC error. Thus, after having chosen a pair $< instruction, symptom >$ to protect, we need to consider the coverage overlap of this instruction and the next one to select.

3. Experimental Results

In this Section, we present the results experiments for two SPECINT2000 benchmarks: `gzip` and `twolf`. For each benchmark we injected 30,000 faults using PIN [3]. A single fault was injected in each run. Our infrastructure determines where to inject the error by first selecting a random dynamic instruction and then flipping a random bit of the output. More sophisticated error injection schemes are possible, but this is part of our future work. For each run we collect a trace with 10,000 entries and let the application run until it finishes. Then, we compare the output of the faulty-run with that of the golden-run as explained in Section 2.

3.1 Application Fault Maskability

Figure 2 shows detailed results of our fault injection experiments. After injecting faults using the techniques described above, we observed that only 2.6% errors in `gzip` and 31.1% errors in `twolf` cause SDC. 62.5% errors in `gzip` and 62.4% errors in `twolf` do not corrupt program output at all, and 34.9% in `gzip` and 6.4% in `twolf` cause abnormal termination.

We consider that an error is derated if it does not manifest in the program output. We define two types of error derating. If the injected error does not manifest in the trace, we call it instruction and control-flow error derating. If the injected error manifests in the trace, but does not manifest in the program output, we call it application derating.

Since our traces only collect the operands of the store intructions and the arguments of system calls such as `printf`, instruction and control-flow error derating can occur because the fault was injected into a dead register or because the fault was masked by one of the instruction not collected in the trace. We do not collect the whole trace (only the first 10,000 entries are collected in the trace after the error is injected), so it is possible for an error to appear in the trace after the 10,000 entries were collected.

Figure 2 shows that instruction and control-flow derating rate (shown as matched traces) is 52.9% for `gzip` and 46.7% for `twolf`. Cook et al. show that the average intruction-level derating rate for the SPECINT2000 benchmarks is 35.9% [1]. This number is smaller than the one we report for two reasons: i) Our numbers include control flow derating, while Cook's don't. In our experiments a

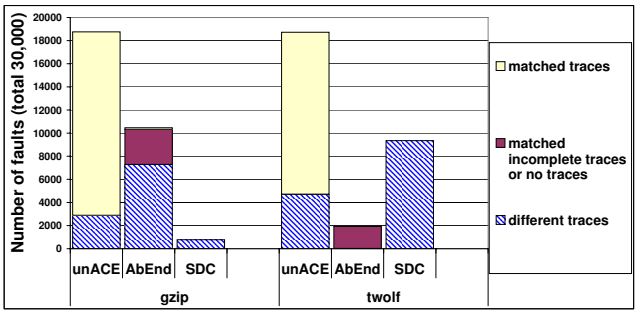

Figure 2. Application Fault Maskability.

branch can take a wrong path, but if memory and system calls are not affected by this error, we count that case in the instruction and control-flow derating rate. ii) We do not inject faults to the instruction opcode, while Cook does. Errors in the instruction opcode usually can not be masked by the instructions.

To compute the application derating we consider that the error is derated if the program output is correct although it generates a faulty entry in the trace. Application derating can occur when two stores write to the same memory location, the first one writes a wrong value and the second one writes a correct one. In `gzip` 9.64% of the errors corrupt the trace but do not corrupt the final program output; for `twolf`, this number is `15.7`%.

3.2 Protection of SDC-indicating intructions

With the information collected in the Symptom Table we compute SDC coverage and SDC-distinguishability for every PC-symptom pair. Then, we set the threshold for SDC-distinguishability to 80% to find PC-symptom pairs that are above the threshold. With this information we can determine which variables need to be protected to avoid SDCs. This process has been done manually and because of that we only selected a few variables in each program. Next, we report the results for `gzip` and `twolf`.

3.2.1 gzip

We call output arguments to the store operands or system call arguments that we collect in the trace. In `gzip` we counted a total of 342 different output arguments in the whole program. Among them, 261 have symptoms (wrong value, wrong control flow, or both) at least during one run, and 81 do not show any symptom. Out of the 261 output arguments that have symptoms, only 145 exhibit symptoms for SDC errors. After setting the threshold for SDC-distinguishability at 80%, we find a total of 101 PC-symptom, with 30 pairs being value based and 71 being control-flow based.

Since the control-flow symptoms account for a larger fraction we searched for other internal variables (variables

that are not stored or passed as system call arguments) that could result in the control flow taking the wrong path and leading to the wrong control flow symptom detected in our experiments. We found that 339 SDC errors propagate to the return value of function $ct\_tally()$, which makes the call to $flush\_block()$ function in a wrong time. We profiled `gzip` and found that $ct\_tally()$ should always return a boolean value, 1 or 0. Thus, a very simple range detector for the return value on the caller side can detect most of these errors when the return value is found to be different from 1 or 0. This simple check detects 328 SDC-errors and there are no false positives at all. We obtain 42.3% SDC-coverage, 100% SDC-distinguishability, and almost zero protection-cost. To increase SDC-coverage we need to relax the threshold for SDC-distinguishability, but we did not do this experiment.

3.2.2 twolf

In `twolf` we counted 1342 output arguments through out the whole program. Among them, 682 have symptoms. Out of these 682 arguments, 635 exhibit symptoms for SDC-errors. After setting the threshold for SDC-distinguishability we chose 1194 PC-symptom pairs, with 1005 being value-based symptom pairs and 189 being control-flow based symptom pairs.

Since the value-based symptom account for a larger fraction, we randomly chose three variable-symptom pair which have relatively high SDC-coverage and high SDC-distinguishability, as shown in Table 2. As the table shows, a perfect value predictor could detect around 90% SDC-errors. However, such a perfect value predictor is not easy to find. In our experiment, we place a very simple range detector for each of the variables. The range detector monitors two properties for a variable: the value of the variable, and the stride from the previous value to the current value of this variable. If the value or the stride violates the profiled range, the range detector will trigger an alarm. The performance overhead for such a range detector on these three variables is 1.3% over the non-protected program. After inserting the simple range detectors and inject the same 30,000 faults, we find that we can detect 3928 SDC-errors (42.0% SDC-coverage), 2 segfault-errors and only 580 correct-errors (87.1% SDC-distinguishability). Notice that there are 4716 unACEs having mismatched value traces (see Figure 2) that would cause false positives, but we avoided most of them.

variable	function	SDC coverage	SDC distinguishability
var1	$new\_dbox()$	90.6%	93.8%
var2	$new\_dbox\_a()$	94.3%	88.6%
var3	$new\_dbox\_a()$	88.5%	92.0%

Table 2. Three chosen variables to protect in twolf

4. Related Work

We compare this work to previous work along two dimensions, fault maskability and invariance-based fault detection.

4.1 Fault Maskability

When the hardware is affected by alpha particles or energy neutrons, the fault may be masked at the micro-architecture level, architecture level, instruction level or application level. Mukherjee et. al [4] and Li et. al [2] study the error mask probability on some structures at the architectural level and use an experimental approach to predict the the probability of the architecture derating an error. Several works have proposed to ignore errors that will be masked by the architecture or the instructions. Weaver et. al [10] modifies the error reporting mechanism to not report detected errors that only affect dynamically dead instructions. Cook et. al [1] observe six categories of instruction level error derating and proposes a mechanism to avoid unnecessary error recovery when a fault is masked by the instructions in the context of a dual modular redundancy (DMR). We focus on instruction and control-flow level and application level fault masking and propose an intelligent fault detector that is able to only detect errors that are not going to be masked by instructions, control flow or the application.

4.2 Invariance-based Fault Detection

The invariance-based fault detection strategy is based on the observation that some variables only take a few values or assume a sequence of a predefined pattern. Thus, the program can be modified to detect when these variables assume values that do not conform to the predicted pattern. Strategies have been proposed to use detectors that detect pattern violations to find transient errors. Racunas et. al [6] use value perturbation screeners to detect hardware transient errors. Sahoo et. al [9] use profile information variables that have a likely invariance property to detect hardware permanent errors. This approach has two disadvantages: i) Although deviation from the invariant is likely to detect an error, this error is only guaranteed to have a local effect and and it is not known if the detected error can later be masked by subsequent instructions in the application. ii) Some errors cannot be detected by simply monitoring invariants.

Our approach is to first try to determine the critical variables, independently on whether they are invariants or not, and then try to find the most efficient method to protect them. This method can be to detect if the variable conforms to the expected pattern of values, but also instruction-level replication or others. Our mechanism can also target spe-

cific types of errors, such as SDCs, while the detection of invariance targets all types of errors.

5. Future Work

Although our preliminary results seem to indicate that our intuition was right, and that only a few variables are important to detect most of the SDC errors, the approach needs to be extended in various directions:

- To accurately model soft errors one should use a HDL simulator and inject faults to latches, buses, combinational logic and SRAM cells, among others. As was the case with faults injected into registers and status flags, some faults will be masked and others will manifest as errors [8]. We are modeling only those errors that appear at the architectural state, but our fault model does not inject errors that corrupt the program counter or instruction opcode. In addition, the fault distribution of a more precise model can be different from the one simulated in our experiments. However, one advantage of our system is that because the error is injected in the binary file, we can inject a large number of errors, collect long traces and let the program run until the end. Due to the long running time, it is unfeasible to run these experiments in an HDL simulator. Thus, a compromise between these two fault injection methodologies is probably the best solution to this problem.

- Currently the process of fault injection and collection of statistics is fully automated; however, the process of protection has been done manually. While it is unclear if this process can be automated, we certainly can provide tools to make this phase at least semi-automatic.

- The approach has been used to detect SDCs, but the same approach can be used to detect segfaults or un-ACE errors. However, we have not run experiments to prove this. Similarly, although our target for this paper were transient errors, the same approach could be used to detect hardware permanent errors. However, in this case the re-execution needs to be done in different hardware to determine if the cause of the error is a defect in the hardware.

- Our experiments have been done using always the same input file. However, we need to evaluate if the results change when using different input files, specially when the application is data dependent. Also, a larger set of applications needs to be evaluated.

Finally, note that this approach has the drawback of not guaranteeing full coverage, although we believe it may obtain high degree of error coverage, with very little performance degradation.

6. Conclusions

In this paper we discuss our approach to find errors that are likely to cause SDC. Our results show that we can detect more than 40% SDC errors with little performance overhead (negligible overhead for `gzip` and 1.3% overhead for `twolf`). Our future work includes applying this approach to a larger set of applications and investigating more sophisticated error-injection techniques.

Acknowledgment

This work was supported by the National Science Foundation under the CSR-AES awards 0615273 and 0509432.

References

[1] J. J. Cook and C. Zilles. A characterization of instruction-level error derating and its implications for error detection. In *DSN*, 2008.

[2] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. Softarch: An architecture level tool for modeling and analyzing soft errors. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 496–505, Washington, DC, USA, 2005. IEEE Computer Society.

[3] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the Intenational Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[4] S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *MICRO*, 2003.

[5] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of International Symposium on Computer Architecture*, pages 99–110, Washington, DC, USA, 2002. IEEE Computer Society.

[6] P. Racunas, K. Constantinides, S. Manne, and S. Mukherjee. Perturbation-based fault screening. *HPCA*, 2007.

[7] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software Implemented Fault Tolerance. In *CGO*, 2005.

[8] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer. An experimental study of soft errors in microprocessors. *IEEE Micro*, 25(6):30–39, 2005.

[9] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. V.Adve, V. S. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *DSN*, 2008.

[10] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In *Proc. of the International Symposium on Computer Architecture*, page 264, 2004.

[11] J. Yu, M. Garzaran, and M. Snir. EsoftCheck: Removal of Non-vital Checks for Fault Tolerance. In *Proc. of CGO*, 2009.