Formal Diagnosis of Hardware Transient Errors in Programs

Layali Rashid, Karthik Pattabiraman and Sathish Gopalakrishnan
The University of British Columbia, Canada
{lrashid, karthikp, sathish}ece.ubc.ca

***Abstract*—As silicon technology continues to scale down and validation expenses continue to increase, more processors with vulnerable parts are shipped to customers. Comprehensive information about architectural units with relatively high failure rates is a critical aspect of the feedback to thread scheduling algorithms and to fault detection and recovery mechanisms.**

We present a technique to identify instructions that cause program failure by utilizing the failure symptoms such as program crash and failing detectors. Our technique employs formal verification and is unique in that it does not require hardware support or special instrumentation of the code. We find that careful engineering of the program's error detectors and their locations highly increase the chances of diagnosing soft errors. We further show that we can diagnose up to 80% of faults using 1-4 fault-detectors for two applications.

I. INTRODUCTION

The large complexity of current processors and the high manpower and design cycle requirements [1-2] of the validation process result in "weak" transistors being shipped to end-users [3]. Such transistors are subject to transient errors (also called single-event-upsets or soft errors). Although the mean time before failure for soft errors in a single processor may be low, systems with tens or hundreds of processors may experience high failure rates (e.g., the Blue-Gene system experiences an error once every 4-6 hours [4]). Further, errors in processors that compute and/or store safety-critical data may lead to catastrophic consequences. Therefore, it is important to identify components that exhibit high rates of transient faults and to schedule tasks around such components or mitigate the effects of the faults.

Transient errors are difficult to diagnose [5] because they (1) often last for only a few cycles, (2) disappear upon re-execution and (3) depend on the processor's physical conditions and the architectural state, which may vary from one execution to another. Moreover, transient errors can propagate in the system and cause failures in components different than their origin [6].

This article focuses on diagnosis of hardware transient errors using formal methods. Formal methods provide formal guarantees on the accuracy of the diagnosis process. Further, such methods can ensure completeness of the diagnoses, i.e., find all faults that could have led to the failure.

In this work, we assume that the error has been detected through runtime error detectors in the program. Error detectors are used to limit error propagation in programs and to ensure fail-stop semantics in the event of an error. Such detectors can be written manually by the programmer [7] or derived automatically through program analysis techniques [8].

The goal of the diagnosis is to track the propagation of transient errors in programs and isolate the first affected instruction. This is the first step in isolating faulty functional units in the processor. Starting from the failure dump, which stores the architected state of the processor at the time of program's failure, we track the error backwards to its point of origin. We use symbolic execution based on model-checking to track the propagation of transient errors in the program. The main advantage of this method over simulation is that it symbolically tracks a whole class of errors simultaneously, which enables rapid exploration of the state space.

Locating the specific instruction that experiences a transient fault is a significant step toward identifying the faulty microarchitectural unit, which would reduce the hardware support needed for diagnosis.

We use model checking to find all transient errors that might go undetected in the presence of a given set of detectors. We build on an existing model-checking framework to determine if the program failure was due to a transient error and if so, which instruction/register originated the error. We utilize the failure dump (i.e., the dump file) to further limit the search-space for diagnosis.

To the best of our knowledge, this is the first paper to propose diagnosis of transient faults using formal techniques and without requiring specialized hardware support. Other diagnosis/debugging techniques (1) use probability models to diagnose faults, and hence do not provide formal guarantees [9], (2) require special hardware support for recording the state of the faulty execution and replaying it [10-11], which significantly adds to the cost of the hardware design, (3) run exhaustive periodic testes using ATPG (Automatic Test Pattern Generation) that incurs substantial overheads (up to 30 seconds) even if the processor is fault-free [12]. Further, methods (2) and (3) are limited to diagnosis of permanent hardware errors and it is not straightforward to extend these techniques for transient errors because transient errors depend on environmental factors and may not be easily reproduced.

The contributions of our work are as follows:

- Proposes a novel technique to formally diagnose soft errors in programs without requiring hardware support.
- Enhances an existing model-checking framework to implement the technique. The enhancements consist of advanced fault models to represent the fault's impact on the program and novel operations to filter solutions[1] based on the required architectural state (at the time of failure).
- Demonstrates the technique for two real applications, namely insertion sort and matrix multiply, both of which are protected by custom error detectors.
- Shows that the technique can diagnose 80% of faults with only 1-4 detectors in the program (for the two programs). It further quantifies the diagnosis accuracy in terms of the detectors' locations and number of detectors in the program.

[1] A solution is a processor state that satisfies the program's model.

II. EXAMPLE

We demonstrate the diagnosis process with a code example (Fig. 1). The code fragment describes a program to compute the factorial of a number read and stored in $1. The program is written in a MIPS-like assembly language. We first explain its operation without errors: The result variable $2 is initialized to 1 (line 1). The loop counter variable $3 is initialized to $1 (line 3) and the loop iterates while the condition ($3 > $4) is true, where $4 is always 1. At each loop iteration, $2 is multiplied by $3 (line 7) and $3 is decremented by one (line 9).

In order to prevent error propagation in the program, we added two detectors to the program (*check* instructions). The first detector asserts that $2 is larger than $1 - 1 while the second asserts that $3 is smaller than the $1 + 1.

Assume that $1 reads the value 5. In a fault-free run, the loop iterates 4 times before it terminates. Also assume that a soft error occurs while executing line 3 such that $3 is 13 instead of 5. As a result of the fault, the program continues execution until it reaches line 10 where detector 2 is triggered (since $3 > $1 + 1). The register file at the time of detection is as follows: ($1 = 5, $2= 13, $3 = 12, $4 = 1, $5 = 1). A crash dump file is created that contains the register file and the number of the triggered detector (i.e., detector 2).

To find the root cause of the fault, we substantially enhances the SymPLFIED [13] framework to perform automated diagnosis of transient faults that affect the program. SymPLFIED symbolically injects a transient fault into each instruction and tests if the fault triggers detector 2. Say SymPLFIED injects a fault into the instruction at line 3, so that $3 stores the value "err" (err represents an erroneous value). When SymPLFIED evaluates line 5 (the loop condition), it forks the execution into two branches, one with $5 = 1 and another with $5 = 0. The branch with $5 = 0 causes the program to print out the result and terminate, while the branch with $5 = 1 enters the loop body. Since line 7 multiplies 1 by err, $2 is updated with err. The instruction Check 1 is forked in a way similar to the loop at line 5, with one branch that continues evaluating the program and another that assumes detector 1 is triggered and stops the execution. However, since SymPLFIED searches for solutions at which detector 2 is triggered, the later branch is dropped. Then SymPLFIED proceeds to evaluate line 9 (for the other branch), which maintains the value err at $3, then triggers detector 2 at line 10 and finds a solution.

Similarly, injecting line 9 with a transient fault triggers detector 2 and reaches another solution. Hence, SymPLFIED finds a correct diagnosis (the former solution) and an additional one (the later solution).

Although manually propagating faults and identifying their root causes is not complex for this example, tracing faults in advanced structures such as nested loops or a faulty address in a memory operation is much more complex. Therefore, we need an automated technique for fault diagnosis.

```
1  movi $2, #1  --- result variable
2  read $1,   --- read from input
3  mov $3, $1    --- loop counter variable = $1
4  movi $4, #1
5 loop:  setgt $5, $3, $4   ---loops while $3 > $4
6  beq  $5, #0, exit
7  mult $2, $2, $3
8  check 1   --- detector 1
9  subi $3, $3, #1
10  check 2   --- detector 2
11  beq  $0, #0, loop   ---loop backedge
12 exit: 12  prints "Factorial = "
13  print $2
--- detectors' definitions
1 $2 > $1 - 1
2 $3 < $1 + 1
```

Fig. 1. A program to compute factorial

III. Diagnosis Approach

In this section, we explain our diagnosis technique and the enhancements we made to SymPLFIED to facilitate this diagnosis. The steps in the diagnosis are as follows:

(1) Store the dump file of the faulty-program when a detector is triggered. The dump file contains the detector that has been triggered by the soft error, the register file contents and the memory contents at the time of the failure.

(2) Extract information from the dump file and use it to formulate a SymPLFIED query. A SymPLFIED query injects a transient fault into the program instructions, one at a time, and compares the architectural state at the moment the program terminates to the one in the query. A match represents a solution and is a diagnosed fault.

To implement the above procedure, we instrumented SymPLFIED with operations that filter solutions based on criteria extracted from the dump file.

For example, if the dump file contains the register file: $1=0, $2=5, $3=-1, $4=8, $5=34, $6=50, $7=21, and detector number 2 has been triggered, then the SymPLFIED query we generate is as follows:

```
search allRegisterErrors(detectors, program)
=>! (S:State) such that (getOutput(S:State)
contains "Exception: Check 2 failed") and
RegisterFile($1 = 0) and RegisterFile($2 = 5)
and RegisterFile($3 = -1) and RegisterFile($4
= 8) and RegisterFile($5 = 34) and
RegisterFile($6 = 50) and RegisterFile($7 =
21) .
```

This query searches for all solutions that trigger the second detector when injecting a fault into the destination register at each instruction, one at a time. The symbol =>! is a built-in Maude [14] operator, it generates all solutions that satisfy the given model and cannot be reduced.

We modified SymPLFIED such that when a detector is triggered, the program execution stops and it prints the message "`Exception: Check x failed`" to the output, where *x* is the detector number. The operation `RegisterFile($1 = 0)` checks if register $1 in the register file found by SymPLFIED is equal to zero. When the program terminates, SymPLFIED checks whether the output contains the phrase "`Exception: Check 2 failed`", and whether the register file matches the one in the query. If SymPLFIED finds a match then it reports a solution. We diagnose the error by extracting the number of the injected-instruction from the solution.

We are conservative in modeling the effects of faults. For example, in a real program an error in a memory-address in a load or store operation either raises a segmentation fault exception or loads incorrect data. We modify the memory-fault model at SymPLFIED such that it loads and stores the value "err" to the destination register if the memory-address is in error. The effect of the conservativeness is that SymPLFIED may incur false-positives; it identifies more faults than one may observe in real runs. However, we believe that completeness of the diagnosis process is more important than incurring a few false-positives.

SymPLFIED has two other fault models: (1) `allMemoryErrors` operation. This operation injects errors into memory locations at each instruction, to model errors that may occur while reading or writing data to memory. (2) `allControlErrors` operation. This operation injects errors into the program-counter at each instruction, to model errors that occur during the fetch stage and result in the wrong instruction being executed. These models are similar to the `allRegisterErrors` operation.

IV. Experimental Methodology

In this section we discuss the steps followed to evaluate our diagnosis approach. The methodology is

as follows (all the following steps are automated using scripts):

(1) We formulate SymPLFIED queries to search for transient fault injections that trigger the program's detectors, one at a time. Therefore, the required number of queries is equal to the number of detectors in the program. For example:

```
search allRegisterErrors(detectors, program)
=>! (S:State) such that (getOutput(S:State)
contains "Exception: Check 1 failed") .
```

This query searches for all solutions that trigger the first detector when injecting a fault into the destination register at each instruction. From each SymPLFIED solution, we extract the injected instruction number, and create a list of instructions that may trigger the specific detector.

(2) To determine if these instructions would trigger a detector in a real run of the application and to collect the dump files, we perform a fault injection campaign using the SimpleScalar simulator [15]. SimpleScalar is a cycle-accurate processor simulator for a MIPS-like instruction set. We use the simulator to inject faults into the program and gather the failure dump in case the fault triggers a detector. Further, SymPLFIED directly supports the instruction set of SimpleScalar and hence we can directly use it for diagnosis.

(3) We scan the dump files and create the appropriate SymPLFIED queries, and then run the queries. Next, we extract the injected instructions from SymPLFIED solutions, and compare each extracted instruction with the corresponding instruction that originally triggered the detector in SimpleScalar. If these match, then we consider the diagnosis a success (otherwise it is 'undiagnosed').

In the experiments, we run each query for at most 5 minutes. The analyses are conducted offline and hence running time is not an overwhelming concern. Further, queries may be parallelized to reduce the time overhead [13].

V. Results

In this section, we report the results of the experiments described in Section 4 for two programs: insertion sort and matrix multiply. Matrix multiply consists of 162 lines of un-commented assembly code - this includes 20 memory instructions, 9 branch instructions and 8 integer-arithmetic instructions. Insertion sort consists of 132 lines of un-commented assembly code, this include 9 memory instructions, 6 branch instructions and 15 integer-arithmetic instructions. The rest of the instructions comprise of assignments, detectors, I/O instructions and function calls. We derive detectors in each program that are based on its functionality, and at the same time have the best possible error coverage [16]. We choose detectors such that they check the correctness of variables (1) with high fanins (fanin here is a dynamic instruction that affects the detector variable) and (2) with long lifetime, which is the distance in time between the variable initialization to its last use.

We study the effect of increasing the number of detectors on the diagnosis accuracy for each program. Tables 1 and 2 show the variation in fault injection/detection rates (with SimpleScalar) and fault diagnosis accuracy (with our technique) with increasing number of detectors.

TABLE I
DIAGNOSIS RESULTS FOR INSERTION SORT

Number of Detectors	1	4	7
Number of faults injected in SS	11	165	198
Number of faults detected in SS	8	64	83
Diagnosed Faults (%)	100	87	89
Undiagnosed Faults (%)	0	13	11

TABLE II
DIAGNOSIS RESULTS FOR MATRIX MULTIPLY

Number of Detectors	1	4	6
Number of faults injected in SS	167	275	286
Number of faults detected in SS	74	135	150
Diagnosed Faults (%)	100	77	80
Undiagnosed Faults (%)	0	23	20

The results from the tables are summarized as follows:

- The number of faults injected in SimpleScalar is proportional to the number of detectors. Recall from the previous section that the first step in our experimental methodology is to find the list of instructions that trigger any detector using SymPLFIED. We inject faults into this list of instructions.
- The percentage of faults detected by the detectors is 50.5 % (on average), for both programs. This is because of the detectors' limitations. For example, a detector that checks if a variable is below a loose threshold (e.g 100), may not be triggered by a small deviation in the variable. The accuracy of detection is orthogonal to the diagnosis accuracy.
- Overall, the proposed technique diagnoses up to 89% of the detected faults for the insertion sort program and up to 80% of the faults for the matrix multiply program.
- Although the percentage of diagnosed faults for both programs decreases when the number of detectors increases from 1 to 4, the absolute number of the diagnosed faults is higher for the 4 detector case. This shows that adding more detectors increases the diagnosis accuracy.

- The undiagnosed faults in both benchmarks are those that (1) affect counters in three-level nested loops and (2) those that affect the detector itself. We believe that these cases are implementation artifacts of the SymPLFIED tool and we plan to address them in future work.
- A single detector in the matrix multiply program diagnoses more faults than four detectors in the insertion sort program. This is due to the choice of the detector's variables. The single detector in the matrix multiply program checks a critical variable (i.e., a variable used by many instructions), while detectors in the insertion sort program check variables that are each used by at most two instructions. This is also why increasing the number of detectors does not improve the detection rate for the matrix multiply program as much as it does for the insertion sort program.

Increasing the number of detectors makes the diagnosis process more deterministic as the detectors reduce the number of candidate solutions (queries with fewer false positives). Note that each detector can be triggered by multiple scenarios of fault injections, and these scenarios increase as the fanin of the detector variable increases, hence there are multiple solutions that SymPLFIED generates for queries augmented with detector number only. We expect that the diagnosis-queries with detector number and register-file contents can generate fewer solutions and hence the diagnosis process will be more deterministic.

VI. Related Work

We classify related diagnosis work into four broad areas: (1) hardware-based techniques, (2) probabilistic techniques, (3) formal methods and (4) periodic-testing techniques.

(1) Hardware-based techniques. Li et al. [10] and Bower et al [11] propose techniques to diagnose hard faults using significant hardware support. This incurs high power overheads and hardware complexity. In contrast, we do not require hardware support for diagnosis. Further, we focus on transient errors which are more difficult to diagnose since they disappear upon re-execution. Park and Mitra [17] use special hardware recorders to collect traces of data and control flows to localize bugs during the post-silicon validation phase. Our technique complements their work by localizing the failure-causing instruction using software methods, as compared to hardware records in the original technique.

(2) Probabilistic techniques. Jha et al. [9], Racunas et al. [18] and Wang and Patel [19] use probabilistic techniques to detect and diagnose errors. These techniques are based on heuristics or probability models and do not take into account for all possible errors, which may result in them missing important cases. In addition, they require a model of the system's error-free behavior which they derive through online learning. Therefore, they may misclassify correct executions as erroneous ones. We do not suffer from this problem as ours is a posteriori technique that derives the correct behavior of the system based on the program's semantics.

(3) Formal methods. Formal methods have been extensively used in diagnosing errors in distributed systems [20-21]. These diagnosis approaches are application-generic and do not take into account the structure/behavior of the program under consideration. Hence, they may incur high rates of false-positives. Further, they model software as a black-box and are hence not fine-grained enough to isolate failures of individual instructions (this is important to identify the fault).

(4) Periodic-testing techniques. Periodic testing techniques [12] provide precise diagnosis for hard errors, however, they require high resource overhead even for fault-free cores. Further, it is not easy to extend these techniques for soft errors since soft errors are non-deterministic and hence difficult to reproduce during testing.

VII. Conclusions and Future Work

We propose a formal technique to diagnose program instructions that are affected by transient errors (as the first step to diagnose the vulnerable functional units) that trigger detectors, with no hardware support. Our diagnosis method is able to diagnose 77%-100% of faults using 1-4 detectors in software.

We propose to conduct this diagnosis technique offline; so as to learn about the fault-prone microarchitectural units and maintain a database with these units, in addition to a table that contains information about common failures and possible sources. This database can provide very valuable feedback to scheduling polices. Further, it can complement detection and recovery methods by providing hints about which units are likely to cause faults and hence reduce the overall detection and recovery overhead.

Based on the results of the study, we believe that software diagnosis of hardware faults is feasible, efficient and can be automated using formal techniques. We will further evaluate this hypothesis by

(1) studying the effects of the detector-variable on the diagnosis process, (2) integrating the register file and the memory contents in the diagnosis process, (3) diagnosing permanent and intermittent faults and (4) improving the scalability of our formal methods by using heuristics.

References

1. Abramovici, M., et al. *A reconfigurable design-for-debug infrastructure for SOCs*. in *Proceeding of the Conference on Design Automation (DAC)*. 2006.
2. Josephson, D., *The Good, the Bad, and the Ugly of Silicon Debug*, in *Design Automation Conference*. 2006, ACM. p. 3 - 6.
3. Constantinides, K., O. Mutlu, and T. Austin. *Online Design Bug Detection: RTL Analysis, Flexible Mechanisms, and Evaluation*. in *Proceeding of the IEEE/ACM International Symposium on Microarchitecture*. 2008.
4. Michalak, S., et al., *Predicting the Number of Fatal Soft Errors in LosAlamos National Laboratory's ASC Q Supercomputer.* IEEE Transactions on Device and Materials Reliability, 2005. **5**(3).
5. Josephson, D.D. *The manic depression of microprocessor debug*. in *Proceedings of the International Test Conference*. 2002.
6. Blome, J., et al., *A Microarchitectural Analysis of Soft Error Propagation in a ProductionLevel Embedded Microprocessor*, in *In Proceedings of the First Workshop on Architecture Reliability*. 2005.
7. Hiller, M., A. Jhumka, and N. Suri. *On the placement of soft-ware mechanisms for detection of data errors.* in *Proceedings of the International Confonference on Dependable Systems and Networks (DSN)*. 2002.
8. Lyle, G., et al. *An End-to-end Approach for the Automatic Derivation of Application-aware Error Detectors*. in *International Conference on Dependable Systems and Networks*. 2009.
9. Jha, S., et al., *Localizing Transient Faults Using Dynamic Bayesian Networks*, in *IEEE International High Level Design Validation and Test Workshop*. 2009. p. 4.
10. Li, M., et al. *Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults*. in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. 2008.
11. Bower, F.A., D. Sorin, and S. Ozev, *Online Diagnosis of Hard Faults in Microprocessors.* ACM Transactions on Architecture and Code Optimization, 2007. **4**(2).
12. Li, Y., S. Makar, and S. Mitra. *CASP: concurrent autonomous chip self-test using stored test patterns*. in *Proceedings of the conference on Design, automation and test in Europe*. 2008.
13. Pattabiraman, K., et al., *SymPLFIED: Symbolic Program-level Fault Injection and Error Detection Framework*, in *Proceeding of the International Conference on Dependable Systems and Networks (DSN)*. 2008.
14. Clavel, M., et al., *Principles of Maude*, in *Proc. First Int'l Workshop on Rewriting Logic and Its Ap-plications*. 1996.
15. Burger, D. and T.M. Austin, *The SimpleScalar tool set, ver-sion 2.0.* Computer Architecture News, 1997. **25**(3).
16. Pattabiraman, K., Z. Kalbarczyk, and R.K. Iyer. *Application-Based Metrics for Strategic Placement of Detectors*. in *Pacific Rim International Symposium on Dependable Computing*. 2005.
17. Park, S.-B. and S. Mitra, *IFRA: Instruction Footprint Recording and Analysis for Post-Silicon Bug Localization in Processors.* Communications of the ACM, 2010. **53**(2).
18. Racunas, P., et al. *Perturbation-Based Fault Screening*. in *Proceedings of the International Symposium on High Performance Computer Architecture*. 2002.
19. Wang, N. and S. Patel, *ReStore: Symptom Based Soft Error Detection in Microprocessors.* IEEE Transactions on Dependable and Secure Computing, 2006. **3**(3).
20. Diaz, M., et al., *Observer-a concept for formal on-line validation of distributed systems.* IEEE Transactions on Software Engineering, 1994. **20**(12): p. 900 - 913
21. Krishnamachari, B. and S. Iyengar, *Distributed Bayesian algorithms for fault-tolerant event region detection in wireless sensor networks.* IEEE Transactions on Computers, 2004. **53**(3): p. 241-250.