DIEBA: Diagnosing Intermittent Errors by Backtracing Application Failures

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

***Abstract*—Intermittent hardware faults have emerged as a leading cause of system failures in the real world. Unlike transient faults, intermittent faults recur at the same location, and need to be diagnosed in order to mitigate their effects. However, unlike permanent faults, intermittent faults are non-deterministic, which makes them challenging to diagnose through traditional methods. We propose a software-based technique to Diagnose Intermittent hardware Errors in microprocessors by Backtracing Application state at the time of a failure (DIEBA). We focus on faults that occur in the micro-architectural units in a core, and either result in program failure or have been detected by software/hardware detectors. We have evaluated DIEBA through fault-injection experiments, and show that it can successfully diagnose 70% of the errors that result in failures or detections.**

I. Introduction

Hardware faults are increasing in microprocessors due to technology scaling and diminishing design margins [11], [3]. Hardware faults can be masked through the use of guard-banding or Dual Modular Redundancy (DMR). Unfortunately, these solutions have significant power overheads, and are untenable as power considerations dominate processor design.

Hardware faults may be broadly classified into transient, intermittent and permanent. Transient faults are typically one-time events that do not recur, while permanent faults persist at the same location indefinitely. Intermittent faults lie between these two extremes, in that they recur at the same location, but do so non-deterministically based on external factors (e.g., temperature) [6]. A recent study has found that about 40% of real-world failures in a processor are caused by intermittent faults [12].

Fault diagnosis is often a precursor to fault recovery. For transient faults, recovery consists of rolling back to a checkpoint and restarting the program's execution. No diagnosis is required for such faults as the fault is unlikely to be encountered again. For permanent faults, the faulty component may need to be disabled during system reconfiguration. Permanent faults can be diagnosed by running additional tests, as the fault is always present at that location.

Intermittent faults, on the other hand, present unique challenges for diagnosis. Unlike permanent faults, they are not deterministic and hence may not appear during testing. Further, they need diagnosis, unlike transient faults, as they are likely to recur at the same location and cause the system to fail. Finally, the diagnosis should incur minimum power and area overheads during fault-free operation. Existing diagnosis techniques either incur considerable energy overheads and design complexity [7] or require that the fault appears deterministically [10] (which does not hold for intermittent faults). Further, many of the techniques require hardware support [13].

Recent studies have proposed to mitigate the effect of permanent faults by fine-grained core reconfiguration around the error-prone microarchitectural unit or pipeline stage [16]. The goal is to keep using other functional parts of the core instead of shutting down the entire core. These studies assume that there exist some techniques to isolate such defective units. Although diagnosis of permanent faults is well explored, intermittent fault diagnosis remains a challenge.

The central question we explore is: *"Can one develop low-overhead, software-based diagnosis methods for intermittent errors to identify faulty functional units in a processor?"* Towards answering this question, our contributions are:

- Introduce a software-based diagnosis technique, DIEBA, that starts from the failure dump of a program (due to an intermittent error) and identifies the faulty micro-architectural unit. DIEBA requires no hardware support, and can be executed on a different core than the one that experienced the error (thus diagnosis does not pause the core's execution after the failure) (Section II).
- Evaluate the accuracy of DIEBA through the use of a cycle-accurate simulator, and through fault-injection experiments based on a micro-architectural model of intermittent faults. We find that DIEBA accurately diagnoses 70% of errors in three functional units of the processor. These units comprise 57.1% of the total area of OpenSPARC-T1 core [15] and 51.4% of POWER4-like processor [9] excluding caches (Section V).

DIEBA uses information about the application's binary code to reconstruct its execution prior to the failure, and attempts to isolate the faulty unit based on error propagation paths in the application. Because DIEBA requires neither online monitoring of the application[1] nor additional testing on the faulty core, it incurs low power and performance overheads during fault-free operation. *To our knowledge, DIEBA is the first approach to diagnose intermittent faults without running additional tests or requiring any hardware support.*

[1]However, it does require a small amount of logging to be performed by the application in software - this is explained later (Section II).

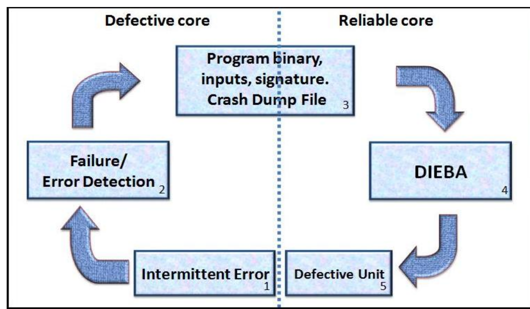


Fig. 1. Overview of our diagnosis technique.

II. Approach

This section presents DIEBA, our approach for diagnosing intermittent hardware errors based on application failures. We start with an overview of DIEBA (Section II-A), describe the construction of the Dynamic Dependence Graph (DDG) [1], which is an essential structure used by DIEBA, then we discuss the design choices and assumptions made in DIEBA's design (Section II-B).

A. Overview

DIEBA is triggered by an intermittent-error detection or program failure; the technique uses program information at the time of error detection or failure, along with some additional information, to identify faulty functional units (Figure 1).

DIEBA's Inputs (Figure 1, Box 3) are: (1) program binary, (2) inputs that were provided to the program including any sources of non-determinism introduced by the operating system, (3) the crash dump file of the program consisting of the register file contents, memory dump and the program and instruction counters[2] (these are available in standard core-dump files), and (4) program signature (see Section III-A2).

DIEBA's Output (Figure 1, Box 5) is the functional unit where the intermittent error occurred. In the current version of DIEBA we support arithmetic and logical units (both integer and floating-point), in addition to the load-store unit. For most fine-grained repair and reconfiguration techniques (e.g., Stagenet [8], Core Cannibalization [16]) it is sufficient to resolve the fault to the granularity of functional units/pipeline stages.

Technique At a high level, DIEBA attempts to reconstruct the execution of the program prior to the error detection or failure. Because we have found experimentally that most intermittent errors lead to program crashes within a few thousands of instructions from the start of an error (Section V), it suffices to reconstruct the last few thousands of instructions of the program prior to the failure (the number of instructions is a configurable parameter of the technique). Based on the reconstruction of the program execution, DIEBA can infer the portions of a program's state that were corrupted by the error (by computing a diff of the two states). DIEBA then backtraces the corrupted data by following the program's dynamic dependence graph (DDG), which is a representation of the dependencies among the program's dynamic instructions to identify the error-propagation paths in the program.

To build a DDG of a program, DIEBA needs two aspects of a program's execution to be recorded. First, DIEBA needs a log of all sources of non-determinism in the program (including its inputs) in order to faithfully reproduce its behaviour. Second, DIEBA needs to log the control flow of the program prior to its failure, because an intermittent error can modify the program's control flow. This is achieved by maintaining a program signature. Based on these two elements, DIEBA reconstructs the program's execution on a different (reliable) core to generate what we call a *fault-free run* (the means of obtaining a reliable core are explained in Section II-B).

Finally, DIEBA attempts to identify the faulty functional unit based on data propagation paths that were affected by the error. This is based on mapping instructions to the corresponding functional units in the processor.

[2] The instruction counter keeps track of the number of dynamic instructions executed in the program, and is available as a performance counter in x86 architectures.

B. Design Choices and Assumptions

a) Design choices: DIEBA is executed on a different core than the one that experienced failure. This is because we do not want the diagnosis process to be affected by the intermittent fault that could potentially recur. Further, because the fault is intermittent, the fault-prone core can continue to be used while the diagnosis is taking place on another core. Therefore, DIEBA does not perturb the fault-prone core.

We assume the availability of a non-faulty core to run DIEBA. One way to achieve this is to run two cores (dual modular redundancy, DMR), so that an error in one core will be caught by the other, as long as both cores do not experience the same error. This does *not incur additional performance and power overheads in the fault-free case*. We assume, as Li et al. [10] have done, that only in the infrequent event of a crash or error detection is the DMR mode invoked [3].

b) Assumptions: In addition to assuming the availability of non-faulty cores, the main assumptions in DIEBA are:

No simultaneous faults We also assume that at most one functional unit of the processor is fault-prone during a given execution of the program. This assumption is justified because intermittent faults are relatively infrequent compared to the typical execution time of many applications, and hence are unlikely to affect multiple functional units.

Fault locations We only consider faults in the processor's load-store unit, the integer arithmetic and logic unit (ALU) and the floating point unit (FPU) and not in the units comprising the processor's front-end such as the fetch and decode units. As a software-based technique, DIEBA does not have enough observability to diagnose errors in such units. Moreover, we assume that all the memory is protected using parity or ECC, and hence does not experience software-visible errors.

[3] It is also possible to run DIEBA at a completely different location by sending the system state at failure over a network connection.

III. DIEBA Implementation

We start this section by explaining how we reconstruct the program's execution prior to the failure. This is a crucial step for the DIEBA technique. Next, we discuss the details of DIEBA's implementation (Section III-B). Then we show an example to illustrate DIEBA's operation (Section III-C).

A. Reconstruction of Program's Execution

As mentioned earlier, reconstructing the execution of the program prior to its failure imposes two monitoring requirements: first, we need to identify and reproduce faithfully all sources of non-determinism (e.g., user inputs and interrupts). Second, we need to capture the control flow prior to the failure. We address these two aspects below.

1) Non-determinism: To enable an execution replay of a program, all sources of non-determinism in the program should be recorded by logging the associated events during the original execution (i.e., record phase). These events include inputs, interrupts, messages exchanged with other processors and inter-leavings among threads (for multi-threaded programs).

There has been significant work in performing deterministic replay in multi-processor and multi-core systems to support software debugging and fault-tolerance. Hardware-based replay systems such as Flight Data Recorder [17], for example, has low performance overheads (less than 2%). Our technique is orthogonal to the specific type of replay system used.

2) Control flow: Intermittent faults can affect branch or jump instructions in a program, in which case the control flow of the affected program will deviate from the correct one. However, we need to capture the control flow of the program prior to its failure in order to reconstruct its execution after the failure. We therefore instrument the program's binary to record a log of the last n basic blocks (BBs). This instrumentation is added to the program's executable, and does not require its source code.

The idea of program signatures is not new. For example, signatures have been extensively used to detect control-flow errors [2]. We use signatures not to detect errors, but rather to replay a program's control-flow.

We partition a program into a set of BBs, and add a store instruction at the end of each BB to save the BB identifier in a buffer. The BB identifier is a unique identifier for each BB in the program.

BB identifiers are stored in a ring buffer in memory. In case of a failure, this buffer will be written to the crash dump file. This buffer is circular since only the most recent identifiers are needed for the diagnosis technique. This is because most intermittent faults cause programs to crash soon after they occur, so only the last few thousands of instructions and hence, the last few thousands of Control-Flow Instructions (CFIs) will be analyzed.

B. The DIEBA Algorithm

We now describe the details of the DIEBA algorithm. The algorithm attempts to reconstruct the execution of the program based on the information gathered in Section III-A. We call this the *fault-free run* of the program, as it is executed on a non-faulty core (see Section II-B). DIEBA then compares the state of the fault-free run at the failing instruction with the state in the failing run. All differences in state are marked as *strong clues*; we know for certain that these elements were corrupted by the error. The algorithm then attempts to trace back from the strong clues using the program's data dependencies to identify the set of all program elements that may have been corrupted by the error. We call these *weak clues*, as these are based on heuristics, and so we cannot be certain that they were corrupted by the error. Finally, the algorithm maps the corrupted propagation paths to the functional units they used in their execution in order to identify the unit that was likely responsible for the corruption. The detailed algorithm is as follows:

(1) Create a fault-free run of the failed program: In this step, we replay the execution of the failed program on a non-faulty core using the information gathered in Section III-A1) as follows: (1) When a non-deterministic data item is read by the program, this data is substituted with the value recorded by the replay tool, and (2) when the program executes a control-flow instruction (CFI), the target address of this instruction is compared with the value recorded in the program's signature (if an entry exists for the CFI). If the addresses do not match, then the CFI's target in the fault-free run is substituted with the address stored in the signature. This makes the fault-free run mimic the control flow of the failing run. In addition, the mismatched CFI is added to the set of strong clues considered by the technique (as it was corrupted by the fault). The fault-free run is terminated when the instruction address and the dynamic count of instructions match that of the failing run, i.e., at the dynamic instruction that crashed the program.

(2) Capture the trace of the fault-free run and construct the DDG: The PC, instruction type (e.g., branch, add) and operands are logged to a trace file for each instruction. Note that this recording is done during diagnosis time only, and does not affect the running program in any way. The Dynamic Dependence Graph (DDG) is constructed from the trace file.

(3) Find the erroneous registers and memory data in the program: Compare the final register file and the memory state of the fault-free run with the corresponding register file and memory state of the failure run to find the set of mismatched registers and memory locations. These locations are also added to the set of strong clues. For convenience, we use the term strong clue to refer to both the members of the set and to the last dynamic instructions (or nodes) that modify them in the program.

(4) Compute backward propagations sets: Use the DDG constructed in step 2 to find all error propagation paths [4] that result in changes to strong clues. This can be done by traversing the predecessors of the instructions that directly affect the erroneous registers, memory locations and CFIs. Such precise backtracing is possible because we use the

[4] A propagation path is a set of instructions that have data dependencies among them.

dynamic trace (step 3) to construct the DDG, and hence have complete information about the execution.

(5) Prune the propagations paths: In the previous step, some of the weak clues found in propagation paths are marked as erroneous just because they have been used as operands in instructions that generated incorrect results, although these operands were not affected by the fault. DIEBA attempts to find these correct operands by checking if they have contributed to some other correct computation(s) in the program, and if so, eliminates them from the corresponding propagation path. This is a heuristic as it is possible that the error is masked by the other computation, and hence we may miss some erroneous data. Nonetheless, we find that this heuristic works well in practice.

(6) Identify the defective unit: In this step, we identify the functional unit that likely experienced the defect as being faulty, by analyzing the error propagation paths and identifying the functional units used by the instructions in the paths. We consider arithmetic/logic instructions (both integer and floating point) and load-store instructions. We then count the number of paths in which a particular unit is used, by associating a counter with the unit. The unit with the highest usage count across all propagation paths is then labeled as defective. The idea of using a counter to infer the defective unit is similar to that of Bower et al. [4].

C. Example

We illustrate the operation of the DIEBA algorithm in the previous section with the example introduced earlier. Table I shows the assembly code fragment corresponding to the example[5].

TABLE I
AN EXAMPLE SET OF DYNAMIC INSTRUCTIONS USED TO DEMONSTRATE THE DIAGNOSIS ALGORITHM.

Node	Address	Instruction
1	0x40d608	addiu $gp[28],$gp[28],-23296
2	0x40d610	lw $s0[16],0($sp[29])
3	0x40d618	addiu $s1[17],$gp[28],42
4	0x40d620	addiu $sp[29],$sp[29],-40
5	0x40d628	addu $s2[18],$zero[0],$v1[3]
6	0x40d630	sw $s2[18],-32396($gp[28])
7	0x40d638	addiu $a1[5],$t4[12],-24
8	0x40d640	sw $a0[4],-32362($gp[28])
9	0x40d648	addu $a2[6],$a1[5],$v1[3]
	0x40d650	jal 401900
10	0x40d658	addu $a1[5],$s1[17],$s0[16]
11	0x40d660	sw $a1[5],-32400($gp[28])
12	0x40d668	addu $a2[6],$a2[6],$s2[18]
13	0x40d670	addiu $sp[29],$sp[29],-40
	0x40d678	jr $a2[6]

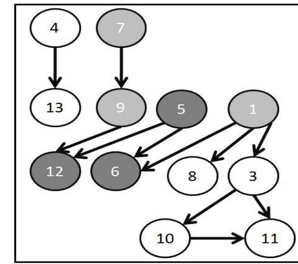


Fig. 2. The DDG constructed using code fragment in Table I. Light grey nodes represent weak clues, while dark grey nodes represent strong clues.

Consider an intermittent fault that affects the Integer ALU, such that three instructions that use the ALU are erroneous; 0x40d628, 0x40d638 and 0x40d648 (nodes 5, 7 and 9). This error causes the jump instruction at 0x40d678 to branch to an invalid code address, which leads to a crash. For this example, we assume that the program is loop-free, and hence there is a one-to-one correspondence between node numbers and addresses in the DDG. Figure 2 shows the DDG corresponding to the code fragment. For simplicity, we have numbered the nodes starting from 1. Also, the DDG only includes the instructions that appear in this example.

We illustrate the algorithm step-by-step on the example.

In step 1, the failed program is re-executed on a fault-free processor, and the fault-free run is constructed. The fault-free run is terminated at PC 0x40d678 as the program crashes at this instruction. For simplicity, we assume that there are no faulty branches or jumps in the replay.

During the re-execution, a trace file is gathered and its DDG is constructed (step 2). The registers contents and the memory state of the fault-free run is compared with that of the failing run (step 3). The technique finds that register #18 is erroneous, because it is updated at PC 0x40d628, which is directly affected by the error. Moreover, register #6 is erroneous because it is also affected directly by the error at PC 0x40d648. Therefore, the set of erroneous registers for this example are #18 and #6. Also, by comparing the memory states of the failure run and the fault-free run, the technique finds that the data that has been stored in memory address -32396($gp[28]) is erroneous (the error propagates to -32396($gp[28]) through register #18 which was affected directly by the error at PC 0x40d628). Thus, the strong clues identified in this example constitute registers #18 and #6 and memory address -32396($gp[28]). These correspond to the DDG nodes: 5, 12 and 6, in dark grey Figure 2.

In step 4, the technique traverses the predecessors of the strong clues (i.e., nodes 5, 12 and 6) and backtraces them until the start of the code fragment (the backtracing window is limited by a few thousands of instructions before the failure location or the start of the program, whichever comes first). The data produced by these instructions are the weak clues. The traversed nodes appear in light grey colour in Figure 2. Table II shows the erroneous propagation paths identified by the technique. For example, nodes 5 and 9 are used to compute node 12. There are two propagation paths corresponding to this node (rows 2 and 3 of the table). Note that immediate values are not represented by nodes in DDG, and hence do not appear in the table. Due to the simplicity of the example, we cannot show the pruning of the propagation paths (step 5).

The final step in the diagnosis (step 6) is to identify the defective unit. We map each propagation path that appears in

[5]The example is in MIPS-like assembly language.

TABLE II
BACKTRACKING PATHS FOR THE EXAMPLE CODE.

Propagation Path in Nodes	Units Used
5	Integer ALU
1 and 5 → 6	Integer ALU and LSU
5 and 9 → 12	Integer ALU
7 → 9	Integer ALU

Table II to the functional units used by the path's instructions. DIEBA maintains a counter for each unit based on the number of paths in which it is used. The Integer ALU's counter will be 4 (used in 4 paths) and the Load-Store Unit's (LSU's) counter will be 1 (used in one path). Therefore, DIEBA concludes that Integer ALU is the defective unit in this example (which is the correct diagnosis).

IV. EXPERIMENTAL SETUP

We use fault-injection to evaluate the accuracy of DIEBA. We built our fault-injection tool based on the cycle-accurate microarchitectural-level SimpleScalar simulator (Alpha sim-outorder) [5]. In our experiment, we assume a simple core with almost no redundancy among functional units. Our modified simulator injects faults into specific microarchitectural units, collects crash dump files and performs program replays.

We use 5 integer and 2 FP benchmarks from the SPEC2006 suite for our evaluation. After the injection, we monitor the benchmark for the following events, (1) the benchmark terminates with a hardware trap, thus leading to a crash, or (2) the number of data values corrupted by the fault crosses a pre-determined threshold (500 in our experiments). The first event signifies a program failure, while the second event signifies a likely detection of the error by a software-based error detection mechanism. This is based on prior work that has shown that errors that have a high fanout in a program often lead to application crashes [14].

The experiment consists of the following two phases.

Injection phase Each fault-injection experiment involves the following parameters: (1) fault location, (2) fault start, (3) fault duration and (4) fault model. We show the possible values/ranges we used in our experiment in Table III. The fault models are chosen to represent common causes of intermittent faults such as wearout and temperature hot-spots. For the integer ALU, multiplier, divider and the floating point (FP) units, we injected the destination register, while for the Load-Store Unit (LSU) we injected the data stored/ loaded and the memory address (each of these locations represents a different injection).

For each benchmark program we injected 3500 faults. Only one fault is injected in each execution to ensure controllability. If the fault results in a program crash or an error is detected (based on the threshold value), then the failure dump consisting of the register file, memory footprint and the program counter is captured at the time of the crash, as also the number of instructions that were executed by the program.

Diagnosis phase We extracted the failure dumps of all injected faults that result in a failure, and ran DIEBA on the extracted dumps. We did not implement execution replay of the program, relying instead on the simulator's capabilities to deterministically reconstruct the program's execution. We also log the control flow of the program internally within the simulator to emulate the recording of program signatures. We compare the output of DIEBA with the injected locations to measure its accuracy.

TABLE III
FAULT-INJECTION PARAMETERS.

Fault Parameter	**Value/Range**
Location-bit	A bit chosen randomly from 0 to 63 in a microarchitectural unit.
Location-unit	Integer ALU, multiplier, divider, LSU (data, read address, write address), FPU
Start cycle	A cycle chosen randomly from 1 to 1,000,000
Duration	5, 50, 100, 500, 10,000 or 20,000 cycle
Model	Stuck-at-one/zero/last-value and Dominant-0/1

V. RESULTS

We present the results of evaluating DIEBA in this section. First, we present the overall results of the fault-injection experiments performed. Next, we measure the accuracy of DIEBA via fault-injections and for different fault locations.

A. Fault Injections

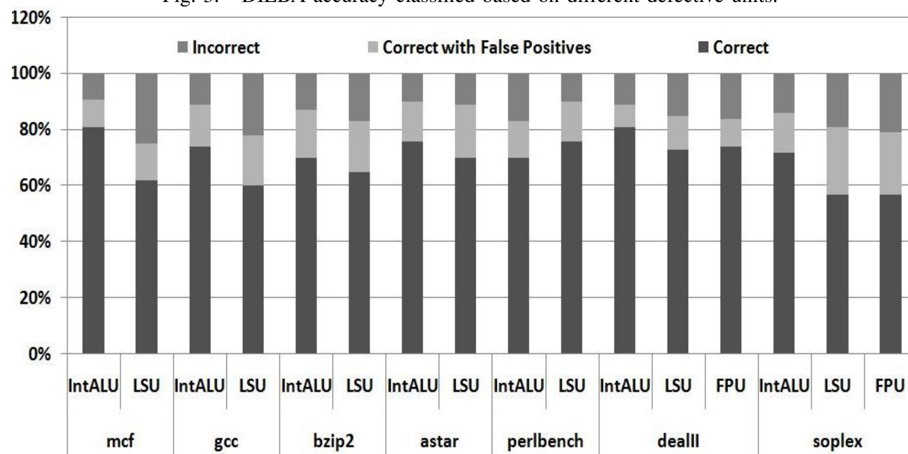
We inject faults into each benchmark program and monitor it for the presence of either crashes or error detections. We find that about 75% of the injected errors are activated (errors manifest themselves to the program). Out of the activated faults, 34% are benign (do not affect the program's state)(on average), 50% cause program crashes, and the remaining 16% lead to Silent-Data Corruptions (SDC). Using our detectors, we are able to detect 6% of the activated faults that cause SDCs. Therefore, of the activated faults, we find that about 56% lead to crashes or detections (TableIV).

TABLE IV
NUMBER/PERCENTAGE OF PROGRAM CRASHES OR ERROR DETECTIONS AND CRASH DISTANCE IN DYNAMIC INSTRUCTIONS (CD) FOR EACH BENCHMARK.

Bench.	mcf	gcc	bzip2	astar	perl.	dealII	soplex
No.	1274	1586	1570	1532	1696	1202	1475
Per.	48%	60%	60%	59%	64%	45%	56%
CD	10235	6365	879	492	371	4512	2319

In order to validate the assumption that programs crash relatively quickly upon encountering an intermittent fault, we measured their crash distance (CD), or the number of dynamic instructions that execute from the injection of the fault to the final crash, for those injections that lead to a crash. We find that the maximum average CD is 10235 (for mcf), and the average CD across all benchmarks is 3688. Thus, software based diagnosis techniques are viable as they only need to log a limited amount of information during program execution.

Fig. 3. DIEBA accuracy classified based on different defective units.



B. DIEBA's accuracy

We run DIEBA when an injected fault causes program crashes or error detections (as per the previous section). DIEBA attempts to identify the functional unit into which the fault was injected. We classify DIEBA's output into three categories: (1) "Correct" where DIEBA uniquely identifies the injected unit, (2) "Correct with false positives" where DIEBA identifies the injected unit along with some other units as possibly defective (recall that we inject only one unit in each run), and (3) "Incorrect" where DIEBA identifies a different unit(s) than the one that was injected.

Figure 3 shows the results of the diagnosis for Integer ALU and LSU for the integer benchmarks, and for the Integer ALU, LSU and FPU for FP benchmarks. The results are expressed as a percentage of the total number of crashes/detections for each benchmark. From Figure 3, we find that *DIEBA successfully diagnoses 85% of the crashes/ detected errors, on average across all benchmarks - this includes the correct (70%) outcome and correct with false-positives outcomes (15%).*

Therefore, DIEBA can diagnose 85% of the intermittent errors that affect three units of the processor. These units comprise nearly 57% of a processor's die area, barring its caches (based on data from the OpenSPARC T1 processor [15]). However, there are portions of the processor that DIEBA does not currently cover, namely the processor front-end and the control-logic units. Extending DIEBA to such components is a subject of future work.

VI. Conclusion

This paper presented DIEBA, a software-only technique for diagnosing intermittent errors based on application failure dumps. Starting from the failure dump, DIEBA is able to isolate the faulty functional unit in the processor that was likely responsible for the failure, and is able to do so without any support from the processor or running any additional tests on it. We show that DIEBA can diagnose 70% of the intermittent errors that affect three units of the processor, with low performance and space overheads.

References

[1] H. Agrawal and J.R. Horgan. Dynamic program slicing. *ACM SIGPLAN Notices*, 25(6):246–256, 1990.

[2] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, and J.A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):627–641, 1999.

[3] Sh. Borkar, T. Karnik, S. Narendra, J. Tschanz, and A. Keshavarzi. Parameter variations and impact on circuits and microarchitecture. *Design Automation Conf.*, pages 338–342, 2003.

[4] F.A. Bower, D. Sorin, and S. Ozev. Online diagnosis of hard faults in microprocessors. *ACM Transactions on Architecture and Code Optimization*, 4(2), 2007.

[5] D. Burger and T.M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, 1997.

[6] C. Constantinescu. Intermittent faults and effects on reliability of integrated circuits. *Reliability and Maintainability Symp.*, pages 370–374, 2008.

[7] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-based online detection of hardware defects: Mechanisms and architectural support and evaluation. *Proc. of the Intl. Symp. on Microarchitecture*, pages 97–108, 2007.

[8] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke. The stagenet fabric for constructing resilient multicore systems. *Intl. Symp. on Microarchitecture*, pages 141 – 151, 2008.

[9] P. Bose J. Srinivasan, S.V. Adve and J.A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. *Intl. Symp. on Computer Architecture*, 33, 2005.

[10] M. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. *Intl. Conf. on Dependable Systems and Networks*, 2008.

[11] J.W. McPherson. Reliability challenges for 45nm and beyond. *ACM IEEE Design Automation Conf.*, pages 176–181, 2006.

[12] E.B. Nightingale, J.R. Douceur, and V. Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer pcs. *European Conf. on Computer Systems*, 2011.

[13] S. Park and S. Mitra. Ifra: Instruction footprint recording and analysis for post-silicon bug localization in processors. *Communications of the ACM*, 53(2), 2010.

[14] K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer. Application-based metrics for strategic placement of detectors. *Pacific Rim Intl. Symposium on Dependable Computing*, pages 75–82, 2005.

[15] A. Pellegrini and V. Bertacco. Application-aware diagnosis of runtime hardware faults. *Intl. Conf. on Computer-Aided Design*, 2010.

[16] B.F. Romanescu and D.J. Sorin. Core cannibalization architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults. *Intl. Conf. on Parallel Architectures and Compiliation*, pages 43–51, 2008.

[17] Min Xu, Rastislav Bodik, and Mark D. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. *Int. symp. on Computer architecture*, pages 122–135, 2003.