Hardware-Software Integrated Diagnosis for Intermittent Hardware Faults

Majid Dadashi, Layali Rashid, Karthik Pattabiraman and Sathish Gopalakrishnan
Department of Electrical and Computer Engineering,
University of British Columbia (UBC), Vancouver
{mdadashi, lrashid, karthikp, sathish}@ece.ubc.ca

***Abstract*—Intermittent hardware faults are hard to diagnose as they occur non-deterministically at the same location. Hardware-only diagnosis techniques incur significant power and area overheads. On the other hand, software-only diagnosis techniques have low power and area overheads, but have limited visibility into many micro-architectural structures and hence cannot diagnose faults in them.**

To overcome these limitations, we propose a hardware-software integrated framework for diagnosing intermittent faults. The hardware part of our framework, called SCRIBE continuously records the resource usage information of every instruction in the processor, and exposes it to the software layer. SCRIBE incurs a performance overhead of 12% and power overhead of 9%, on average. The software part of our framework is called SIED and uses backtracking from the program's crash dump to find the faulty micro-architectural resource. Our technique has an average accuracy of 84% in diagnosing the faulty resource, which in turn enables fine-grained deconfiguration with less than 2% performance loss after deconfiguration.

Keywords: Intermittent Faults, Backtracking, Dynamic Dependence Graphs, Hardware/Software Co-design

I. Introduction

CMOS scaling has exacerbated the unreliability of Silicon devices and made them more susceptible to different kinds of faults [1]. The common kinds of hardware faults are transient and permanent. However, a third category of faults, namely intermittent faults has gained prominence [2]. A recent study of commodity hardware has found that intermittent faults were responsible for at least 39% of computer system failures due to hardware errors [3]. Unlike transient faults, intermittent faults are not one-off events, and occur repeatedly at the same location. However, unlike permanent faults, they appear non-deterministically, and only in certain circumstances.

Diagnosis is an essential operation for a fault-tolerant system. In this paper, we focus on diagnosing intermittent faults that occur in the processor. Intermittent faults are caused by marginal or faulty micro-architectural components, and hence diagnosing such faults is important to isolate the faulty resource [4], [5], [6]. Components can experience intermittent faults either due to design and manufacturing errors, or due to aging and temperature effects that arise in operational settings [2]. Therefore, the diagnosis process should be run throughout the life-time of the processor rather than only at design validation time. This makes it imperative to design a diagnosis scheme that has low online performance and power overheads. Further, to retain high performance after repair, the diagnosis should be fine-grained at the granularity of individual resources in a microprocessor, so that the processor can be deconfigured around the faulty resource after diagnosis [7].

Diagnosis can be carried out in either hardware or software. Hardware-level diagnosis has the advantage that it can be done without software changes. Unfortunately, performing diagnosis entirely in hardware incurs significant power and area overheads, as diagnosis algorithms are often complex and require specialized hardware to implement. On the other hand, software-based diagnosis techniques only incur power and performance overheads during the diagnosis process, and have zero area overheads. Unfortunately, software techniques have limited visibility into many micro-architectural structures (e.g., the reorder buffer) and hence cannot diagnose faults in them. Further, software techniques cannot identify the resources consumed by an instruction as it moves through the pipeline, which is essential for fine-grained diagnosis.

In this paper, we propose a hardware-software integrated technique for diagnosing intermittent hardware errors in multi-core processors. As mentioned above, intermittent faults are non-deterministic and may not be easily reproduced through posteriori testing. Therefore, the hardware portion of our technique continuously records the micro-architectural resources used by an instruction as the instruction moves through the processor's pipeline, and stores this information in a log that is exposed to the software portion of the technique. We call the hardware portion SCRIBE. When the program fails (due to an intermittent fault), the software portion of our technique uses the log to identify which resource of the microprocessor was subject to the intermittent fault that caused the program to fail. The software portion runs on a separate core and uses a combination of deterministic replay and backtracking from the failure point, to identify the faulty component. We call the software portion of our technique SIED, which stands for *Software-based Intermittent Error Diagnosis*. SCRIBE and SIED work in tandem to achieve intermittent fault diagnosis.

Prior work on diagnosis [5] has either assumed the presence of fine-grained checkers such as the DIVA checker [8], or has assumed that the fault occurs deterministically [9], which is true for permanent faults, but not intermittent faults. In contrast, our technique does not require any fine-grained checkers in the processor nor does it rely upon determinism of the fault, making it well suited for intermittent faults. Other papers [10], [11] have proposed diagnosis mechanisms for post-Silicon validation. However, these approaches target

design faults and not operational faults, which is our focus. *To the best of our knowledge, we are the first to propose a general purpose diagnosis mechanism for in-field, intermittent faults in processors, with minimal changes to the hardware*.

The main contributions of the paper are as follows:

i) Enumerate the challenges associated with intermittent fault diagnosis and explain why a hybrid hardware-software scheme is needed for diagnosis.
ii) Propose SCRIBE, an efficient micro-architectural mechanism to record instruction information as it moves through the pipeline, and expose this information to the software layer.
iii) Propose SIED, a software-based diagnosis algorithm that leverages the information provided by SCRIBE to isolate the faulty micro-architectural resource through backtracking from the failure point,
iv) Conduct an end-to-end evaluation of the hybrid approach in terms of diagnosis accuracy using fault injection experiments at the micro-architectural level.
v) Evaluate the performance and power overheads incurred by SCRIBE during fault-free operation. Also, evaluate the overhead incurred by the processor after it is deconfigured upon a successful diagnosis by our approach.

Our experiments on the SPEC2006 benchmarks show that SCRIBE incurs an average performance overhead of 11.5%, and a power consumption overhead of 9.3%, for a medium-width processor. Further, the end-to-end accuracy of diagnosis is 84% on average across different resources of the processor (varies from 71% to 95% depending on the pipeline stage in which the fault occurs). We also show that with such fine-grained diagnosis, only 1.6% performance overhead will be incurred by the processor after deconfiguration, on average.

II. Background

In this section, we first explain what are intermittent faults, and their causes. We then explain why resource level, online diagnosis is needed for multi-core processors. Finally, we explain the Dynamic Dependence Graph (DDG), which is used in our paper for diagnosis.

A. Intermittent faults: Definition and Causes

Definition We define an intermittent fault as one that appears non-deterministically at the same hardware location, and lasts for one or more (but finite number of) clock cycles. The main characteristic of intermittent faults that distinguishes them from transient faults is that they occur repeatedly at the same location, and are caused by an underlying hardware defect rather than a one-time event such as a particle strike. However, unlike permanent faults, intermittent faults appear non-deterministically, and only under certain conditions.

Causes: The major cause of intermittent faults is device wearout, or the tendency of solid-state devices to degrade with time and stress. Wearout can be accelerated by aggressive transistor scaling which makes processors more susceptible to extreme operating condition such as voltage and temperature fluctuations [12], [13]. In-progress wearout faults are often intermittent as they depend on the operating conditions and the circuit inputs. In the long term, such faults may eventually lead to permanent defects. Another cause of intermittent faults is manufacturing defects that escape VLSI testing [14]. Often, deterministic defects are flushed out during such testing and the ones that escape are non-deterministic defects, which emerge as intermittent faults. Finally, design defects can also lead to intermittent faults, especially if the defect is triggered under rare scenarios or conditions [15]. However, we do not consider intermittent faults due to design defects in this paper.

B. Why resource-level, online diagnosis ?

Our goal is to isolate individual micro-architectural resources and units that are responsible for the intermittent fault. Fine-grained diagnosis implicitly assumes that these resources can be deconfigured dynamically in order to prevent the fault from occurring again. Other work has also made similar assumptions [5], [9], [6]. While it may be desirable to go even further and isolate individual circuits or even transistors that are faulty, it is often difficult to perform deconfiguration at that level. Therefore, we confine ourselves to performing diagnosis at the resource level.

Another question that arises in fine-grained diagnosis is why not simply avoid using the faulty core instead of deconfiguring the faulty resource. This would be a simple and cost-effective solution. However, this leads to vastly lower performance in a high-performance multi-core processor, as prior work has shown [7], [6]. Finally, the need for online diagnosis stems from the fact that taking the entire processor or chip offline to perform diagnosis is wasteful, especially as the rate of intermittent faults increases as future trends indicate [14]. Further, taking the chip offline is not feasible for safety-critical systems. Our goal is to perform online diagnosis of intermittent faults.

C. Dynamic Dependency Graphs

A dynamic dependency graph (DDG) is a representation of data flow in a program [16]. It is a directed acyclic graph where graph nodes or vertices represent values produced by dynamic instructions during program execution. In effect, each node corresponds to a dynamic instance of a value-producing program instruction. Dependencies among nodes result in edges in the DDG. In the DDG, there is an edge from node N_1 (corresponding to instruction I_1) to node N_2 (corresponding to instruction I_2), if and only if I_2 reads the value written by I_1 (instructions that do not produce any values correspond to nodes with no outgoing edges).

III. Approach

This section first presents the fault model we consider. It then presents the challenges of intermittent fault diagnosis. Finally, it presents an overview of our approach and how it addresses the challenges.

A. Fault Model

As mentioned in Section II-A, intermittent faults are faults that last for finite number of cycles at the same micro-architectural location. We consider intermittent faults that occur in processors. In particular, we consider faults that

occur in *functional units*, *reorder buffer*, *instruction fetch queue*, *load/store queue* and *reservation station entries*. We assume that caches and register files are protected using ECC or parity and therefore do not experience software visible faults. We also assume that the processor's control logic is immune to errors, as this is a relatively small portion of the chip [17]. Finally, we assume that a component may be affected by at most one intermittent fault at any time, and that the fault affects a single bit in the component (stuck-at zero/one), lasting for several cycles.

B. Challenges

In this section, we outline the challenges that an intermittent fault diagnosis method needs to overcome.

Non-determinism: Since intermittent faults occur non-deterministically, re-execution of a program that has failed as a result of an intermittent fault, often results in a different event sequence than the original execution. In other words, the sequence of events that lead to a failure is not (necessarily) repeatable under intermittent faults.

Overheads: An intermittent fault diagnosis mechanism should incur as low overhead as possible in terms of performance, area and power, especially during fault-free operation, which is likely to be the common case.

Software Layer Visibility: Software diagnosis algorithms suffer from limited visibility into the hardware layer. In other words, software-only approaches are not aware of what resources an instruction has used since being fetched until retiring from the pipeline (the only inferable information from an instruction is the *type* of functional unit it has used).

No information about the faulty instructions: To find the faulty resource, the diagnosis algorithm needs to have information about the instructions that have been affected by the intermittent fault in order that the search domain of the faulty resource can be narrowed down to resources used by these instructions. One way to obtain this information is to log the value of the destination of every instruction at runtime, and to compare its value with that of a fault-free run (more details in Section III-C). However, logging the value of every executed instruction in addition to its resource information can result in prohibitive performance overheads as we show in Section VI. Therefore, we need to infer this information from the failure log instead.

C. Overview of our Approach

In this section, we present an overview of our approach and how it addresses the challenges in section III-B.

We propose a hybrid hardware-software approach for diagnosis of intermittent faults in processors. Our approach consists of two parts. First, we propose a simple, low-overhead, hardware mechanism called SCRIBE to record information about resource usage of each instruction and expose this information to the software. Second, we propose a software technique called SIED that uses the recorded information upon a failure (caused by an intermittent fault) to diagnose the faulty resource by backtracking from the point of failure through the program's DDG (see Section II-C). The intuition is that errors propagate along the DDG edges starting from the instruction that used the faulty resource, and hence backtracking on the DDG can diagnose the fault.

Assumptions: We make the following assumptions about the system:

i) We assume a commodity multi-core system in which all cores are homogeneous, and are able to communicate with each other through a shared address space.

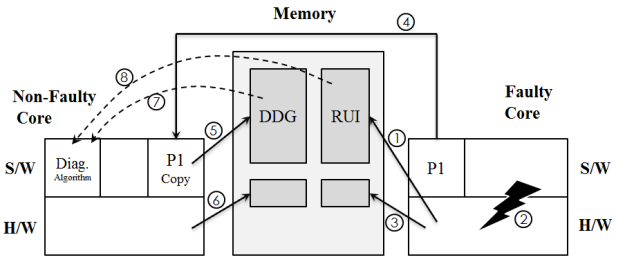
ii) We assume the availability of a fault-free core to perform the diagnosis, e.g. using Dual Modular Redundancy (DMR). This is similar to the assumption made by Li et al. [9]. The fault-free core is only needed during diagnosis.

iii) The processor is able to deterministically replay the failed program's execution. Researchers have proposed the use of deterministic replay techniques for debugging programs on multi-core machines [18], [19]. This is needed to eliminate the effect of non-deterministic events in the program during diagnosis (other than the fault).

iv) The fault has already been identified as an intermittent fault prior to diagnosis. In particular, it has been ruled out to be a transient fault - this can be done by only invoking diagnosis if there are repeated failures. For example, there has been work on distinguishing intermittent faults from transient faults using a threshold mechanism [20].

Steps : Figure 1 shows the sequence of steps our technique would follow to diagnose a fault.

1) As the program executes, the hardware layer SCRIBE logs the Resource Usage Information (RUI) of the instructions (step 1 in Figure 1) to memory. Every instruction has an RUI, which is a bit array indicating the resources it has used while moving through the processor's pipeline. SCRIBE is presented in Section IV.
2) Assume that the program fails as a result of an intermittent fault burst in one of the processor resources (step 2). This failure can occur due to a crash or an error detection by the application (e.g. an assertion failure). The registers and memory state of the application is dumped to memory, typically as a core dump (step 3).
3) The software layer diagnosis process, SIED is started on *another* core. This core is used to perform the diagnosis and is assumed to itself be fault-free during diagnosis (see assumptions). SIED replays the program using deterministic replay mechanisms, and constructs the DDG (steps 4 and 5) of the replayed program. The original program can be resumed on the core that experienced the intermittent fault, as SIED does not interfere with its subsequent execution.
4) When the replayed program reaches the instruction at which the orignal program failed, SIED dumps its register and memory state to memory (step 6).
5) SIED merges the DDG from step 5 with the RUI log in step 1, to build the *augmented DDG*. This is a DDG in which every node contains the RUI of its corresponding instruction in the program.
6) SIED then compares the memory and register states dumped in steps 3 and 6 to identify the set of nodes in the augmented DDG that differ between the original and replayed execution. Because the replayed execution used deterministic replay, any differences between the two executions are due to the intermittent fault. In case of no deviation between two executions, a software bug is diagnosed. This is similar to the diagnosis decision made by Li et al. in [9].



1) Gather RUI and log to memory (SCRIBE)
2) Failure due to intermittent fault
3) Log program's register and memory state (core dump)
4) Deterministic replay on another core (SIED)
5) Construct replayed program's DDG (SIED)
6) Log replayed program's register and memory state (SIED)
7) Construct augmented DDG and backtrack using analysis heuristics (SIED)

Figure 1: End to end scenario of failure diagnosis by SCRIBE and SIED. The steps in the figure are explained in the box.

7) Finally, SIED backtracks from the faulty nodes in the augmented DDG using *analysis heuristics* to find the faulty resource (steps 7 and 8). The details of how SIED works are explained in Section V.

Challenges Addressed: We now illustrate how our technique satisfies the constraints posed in Section III-B.

Non-determinism: Our technique gathers the micro-architectural resource usage information online using the SCRIBE layer (Step 1). Therefore, it requires determinism neither in resource usage nor fault occurrence during the replay.

Overheads: Our technique initiates diagnosis only when a crash or error detection occurs, thus the diagnosis overhead is not incurred during fault-free execution. However, the SCRIBE layer incurs both performance and power overheads as it continuously logs the resource usage information of the instructions executing in the processor. Note that SCRIBE only exposes the hardware RUI information to the software layer. The complex task of figuring out the faulty component is done in software. Hence, the power overhead of SCRIBE is low. We describe the optimizations made to SCRIBE to keep its performance overhead low in Section IV. We present the performance and power overheads in section VI-B.

Software-layer visibility: The SCRIBE layer records the information on micro-architectural resource usage and exposes it to software, thus solving the visibility problem.

No information about faulty instructions: Our technique does not log the destination result of each instruction, and hence cannot tell which instructions have been affected by the fault. Instead, SIED uses the replay run to determine which registers/memory locations are affected by the fault, and backtracks from these in the DDG to identify the faulty resource.

IV. SCRIBE: Hardware Layer

We propose a hybrid diagnosis approach involving both hardware and software. SCRIBE is the hardware part of our hybrid scheme and is responsible for exposing the micro-architectural **R**esource **U**sage **I**nformation (RUI) to the software layer, SIED. This allows SIED to identify the faulty resource(s) upon a failure due to an intermittent fault. In addition, SCRIBE also logs the addresses of the executed branches, so that the program's control flow can be restored in case of a failure (Section V-A). The detailed design of the SCRIBE layer was presented in our earlier work [21].

A. RUI Format

A resource in a superscalar processor consists of the pipeline buffers and functional units. We use the term, Resource Usage Information (RUI) to denote the set of micro-architectural resources used by a single instruction as it moves through the superscalar pipeline. The RUI records the resources used by the instruction in each pipeline stage, as a bitmap. Each field of the RUI corresponds to a single resource class in the pipeline. For example, consider an *add* instruction which is assigned to entry 4 of the Instruction Fetch Queue (IFQ), entry 7 of the Reorder Buffer (ROB), entry 24 of reservation station (RS) and also uses the second integer ALU of the processor (FU). It does not use the Load Store Queue (LSQ), though other instructions may do so and hence space is reserved in the RUI for the LSQ as well. The RUI of this instruction is shown in Figure 2.

The RUI entries are stored in a circular buffer in the process's memory address space as the program executes on the processor. The size of the RUI buffer is determined by the worst-case number of instructions taken by programs to crash or fail after an intermittent fault. Because this number can be large, keeping the buffer on chip would lead to prohibitive area and power overhead. Hence we choose to keep the RUI information in the memory instead of on chip. Therefore, in our case, the buffer size is bounded only by the memory size.

000100	0000111	0011000	0001	111111
IFQ	**ROB**	**RS**	**FU**	**LSQ**

Figure 2: The RUI entry corresponding to an *add* instruction

B. SCRIBE structure

To implement SCRIBE, we augment each Reorder Buffer (ROB) entry with an X bit field ($X \propto lg(\textit{Total number of resources})$) to store the RUI of the instruction corresponding to that entry. This field is filled with a valid RUI entry as the instruction traverses the pipeline and makes use of specific resources. As the instruction has completed its execution when it reaches the commit stage, its complete RUI is known when in the commit stage. The RUI entries are sent to the memory hierarchy when their instructions are retired from ROB, and hence only the RUI entries of the instructions

on the correct path of branch prediction will be sent to the memory.

SCRIBE consists of two units:. **(i)** The *logging unit* is in charge of aligning the RUI entries and sending them to the priority handling unit. **(ii)** The *priority handling unit* is in charge of choosing between a regular store and a logging store to send to memory. We name the process of sending RUI to the memory hierarchy as a *Logging Store*.

Logging Unit: Figure 3 shows the design of the logging unit, consisting of *logging buffer*, *alignment circuit*, and *LogSQ*.

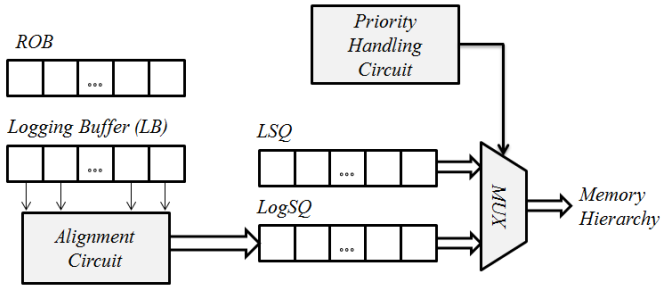


Figure 3: The *Logging Unit* includes the *Logging Buffer*, *Alignment Circuit* and *LogSQ*

When an instruction is retired from the ROB, the RUI field of its ROB entry will be inserted into the Logging Buffer (LB). The LB is a dual partitioned queue and is in charge of keeping the RUI of the retired instructions. Each of the partitions of the *LB* get filled separately. To enable faster writing of the RUI data to memory, we store them as quad-words in memory. The alignment circuits creates quad-words from RUI data in the LB and sends them to the LogSQ. When one of the partitions becomes full, its data is processed by the alignment circuit and the other partition starts getting filled and vice versa. Thus, *data processing* and *filling* modes alternate with each other in each partition of the logging unit.

Logging Store Queue (LogSQ) buffers the quad-words sent by the alignment circuits before they are sent to memory. These quadwords compete with the memory traffic sent by the regular loads and stores of the program. This process is explained below. If the LogSQ is full, the alignment circuits have to be stalled until a free entry in the logSQ becomes available.

Priority Unit: The goal of the priority handling unit is to mediate accesses to main memory between the logging stores and the regular stores performed by the processor. The priority handling unit consists of the priority handling circuit, which makes the decision of which store to send to memory, and a multiplexer to select between the regular store instructions and the *logging store*s.

When both a regular load/store instruction from the processor and a *logging store instruction* from the *logSQ* are ready, one of them has to be chosen to be sent to the memory hierarchy. If logging stores are not sent to the memory on time, the logSQ will become full and the instruction retiring mechanism will stall, thereby degrading performance. On the other hand, if regular stores are not sent to memory in time, the processor's commit mechanism will stall, also degrading performance.

Our solution is to use a hybrid approach where we switch the priorities between the logging stores and the regular stores based on the size of the LogSQ. In other words, we prioritize regular load/store instructions by default, until the logging mechanism starts stalling the commit stage (because of one partition becoming full before the other one is processed). At this point, the logging store instructions gain priority over regular load/stores, until the logSQ is drained.

V. SIED: Software Layer

In this section, we present SIED, the software portion of our technique.

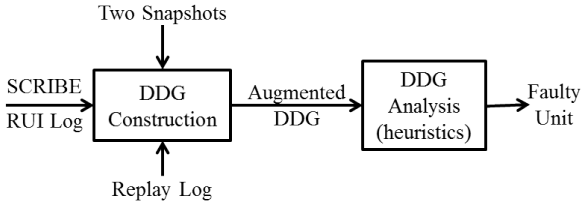


Figure 4: Flow of information during the diagnosis process

SIED is launched as a privileged process by the operating system on a separate core, which enables it to read the RUI segment in the failed program's memory written to by SCRIBE. Therefore, SIED has access to the history of dynamic instructions executed before the failure, and the micro-architectural resources used by those instructions.

Figure 4 shows the steps taken by SIED after a failure. First, the program is replayed on a separate core until the failed instruction, during which its DDG is built. The DDG is augmented with the RUI and the register/memory dumps from the original and replayed program executions. This process is explained in Section V-A. The augmented DDG is then fed to the DDG analysis step in Figure 4 which uses backtracking of DDG to find the candidates of the faulty resource. This process is explained in Section V-B.

Example: We consider the program in Table I as a running example to explain the diagnosis steps. The example is drawn from execution of the benchmark *mcf* from SPEC 2006 benchmark suite on our simulator. However, some instructions have been removed from the real example to illustrate as many cases as possible in a compact way. As the program is executing, SCRIBE monitors the execution of instructions and logs their RUI to memory. The RUI logged by SCRIBE during the original execution is shown in Table I (the real RUI history includes a few thousands of entries; however, we only show the last few entries for brevity). For example, row #2 in Table I shows that the *store quadword* instruction has used entry 26 of the ROB, entry 15 of LSQ, entry 16 of IFQ, entry 11 of RS and functional unit 5 which is one of the memory ports (we consider memory ports as functional units).

Assume that in this example, the processor has multiple functional units, and the second functional unit (fu-1) is

#	Instruction	rob	lsq	ifq	rs	fu
1	addi r1, -1, r1	25	-	15	16	1
2	stq r1, 400(r15)	26	15	16	11	5
3	bic r3, 16, r3	52	-	2	52	2
4	stl r3, 0(r9)	53	24	3	46	5
5	bis r31, r15, r30	84	-	1	7	1
6	ldq r1, 0(r30)	85	2	2	38	6
7	ldq r3, 8(r30)	86	3	3	19	6
8	ldq r30, 16(r30)	87	4	4	40	5
9	stq r5 , -32(r30)	88	5	5	44	6

Table I: RUI of the instructions logged by SCRIBE. The original execution crashes at instruction 9.

experiencing an intermittent fault that is triggered non-deterministically and lasts for several cycles. When the functional unit experiences the fault, one of the bits in its output becomes stuck at $zero$ for this time period. This causes an incorrect value to be produced, as a result of which the program crashes. After the crash, the entire register and memory state of the process is dumped to memory. For this example, we only show the register and memory values produced by the instructions in Table I. These values are shown in Table II, column "*Snapshot Original*". The "producer index" column represents the index of the instructions in Table I that last wrote to the locations in the second column.

Producer Index	Mem/Reg Location	Producer	Snapshot Original	Snapshot Replayed
2	0xd3e0	stq r1, 400(r15)	8	12
4	0xd988	stl r3, 0(r9)	10	10
6	r1	ldq r1, 0(r30)	16	0
7	r3	ldq r3, 8(r30)	8	20
8	r30	ldq r30, 16(r30)	20	0

Table II: Snapshots: These represent the memory and register state dumps after the original and replayed executions

A. DDG Construction with RUI

As mentioned in Section III-C, SIED uses deterministic replay techniques to replay the execution of the failed program and build its DDG. We refer to the first execution leading to the failure as the *original execution* and the second execution performed by SIED as the *replayed execution*.

The steps taken by SIED to build the DDG are as follows (step numbers below correspond to those in Figure 1):

i) The program is started from a previous checkpoint or from the beginning and replayed. However, the replayed program's control-flow may not match the control flow of the original execution, as the latter may have been modified by the intermittent fault. To facilitate fault diagnosis, the only difference between the original and the replayed execution should be the intermittent fault's effects on the registers and memory state. Therefore, the control flow of the replayed execution (target addresses of the *branch* instructions) is modified to match the original execution's control flow (step 4). To obtain the original execution's control flow, SCRIBE logs the branch target addresses of the program in addition to its RUI.

ii) From the replayed execution, the information needed for building the Dynamic Dependence Graph (DDG) of the program is extracted and the DDG is built (step 5). Figure 5 shows the DDG for our example. The information required for building the DDG can be extracted by using a dynamic binary instrumentation tool (e.g. Pin [22]). We note that the overheads added by such tools would only be incurred during failure and subsequent diagnosis, and not during regular operation.

iii) When the program flow of the replayed execution reaches the crash instruction (the instruction at which the original execution crashes), the register and memory state of the replayed execution is dumped to memory (step 6). There could be rare cases in which the replayed execution fails due to inconsistency between the control flow and the data. These cases lead to the diagnosis process being stopped if happened before reaching to the crash instruction. In the example, the replayed execution stops when reaching instruction 9 and the column "*snapshot replayed*" in Table II represents the register and memory state of the replayed program at that instruction.

iv) The snapshots taken after the original and replayed executions are compared with each other to identify the final values that are different from each other. Because we assume a deterministic replay, any deviation in the values must be due to the fault. The producer instructions of these values are marked as *final erroneous* (or *final correct*) if the final values are different (or the same) in the DDG. The branch instructions that needed to be modified in step (i) to make the control flows match are also marked as *final erroneous* in the DDG. In the example, the values in the snapshot columns of Table II are compared, and the differences identified. The nodes corresponding to the instructions creating the mismatched values are marked in the DDG as final erroneous nodes (nodes 2, 6, 7 & 8), while node 4 with matching values, is marked as final correct.

v) The RUI of each instruction is added to its corresponding node in DDG. We call the resulting graph, the *augmented DDG*. The augmented DDG is used to find the faulty resource as shown in the next section.

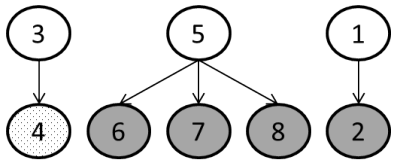


Figure 5: DDG of the program in the running example. Gray nodes are *final erroneous* and the dotted node is *final correct*

B. DDG Analysis

This section explains how SIED analyzes the augmented DDG to find the faulty resource. Because each dynamic instruction corresponds to a DDG node, we use the terms node and instruction interchangeably. The main idea is to start from final erroneous nodes in the augmented DDG (identified in Section V-A), and backtrack to find nodes that have originated the error, i.e., the instructions that have used the faulty resource. The faulty resource is found

by considering the intersection of the resources used by multiple instructions that have originated the errors. Recall that the list of resources used by an instruction is present in its corresponding node in the augmented DDG.

There are three types of nodes in the augmented DDG: **i)** Nodes that have used the faulty resource (originating nodes), **ii)** Nodes to which the error is propagated from an ancestor, **iii)** Nodes that have produced *correct* results (correct nodes). The goal of backtracking is to search for the originating nodes, by going backward from the final erroneous nodes (i.e., erroneous nodes in the final state), while avoiding the correct nodes. Naive backtracking does not avoid correct nodes, and because there can be many correct nodes in the backward slice of a final erroneous node, it will incur false-positives. Therefore, we propose two heuristics to narrow down the search space for the faulty resource based on the following observations:

i) If a final erroneous node has a correct ancestor node, the probability of the originating node being in the path connecting those two nodes is high. In other words, the faulty resource is more likely to be used in this path.
ii) Having a final correct descendent decreases the probability that the node is erroneous.
iii) Having an erroneous ancestor decreases the probability of the node being an originating node.
iv) An erroneous node with all correct predecessors is an originating node.

Heuristics: To find faulty resources, each resource in the processor is assigned a counter which is initialized to $zero$. The counter of a resource is incremented if an instruction using that resource is likely to participate in creating an erroneous value, as determined by the heuristics. Resources having larger counter values are more likely to be faulty.

Algorithm 1 shows the pseudocode for heuristic 1. The main idea behind heuristic 1 is to examine the backward slices of the final erroneous nodes and increase the counter values of the appropriate resources based on the first three observations. In lines 3 to 8, for each final erroneous node n, the set S_{n1} is populated with the nodes between n and its final correct ancestors. The counters of the resources used by the nodes in the S_{n1} are incremented. Lines 9 to 11 correspond to the second observation. Every node in the backward slice of the final erroneous node n is added to set S_{n2} unless it has a final correct descendent. Finally, in lines 12 to 17, the nodes that are added to the set S_{n2} are checked to see if they have a faulty ancestor. If so, their counters are incremented by 0.5, and if not, the counters are incremented by 1. This is in line with the third observation that nodes with faulty ancestors are less likely to be originating nodes.

Algorithm 2 presents the second heuristic which is based on Observation 4. The algorithm starts from the final correct nodes and recursively marks the nodes that are likely to have produced correct output (lines 1 to 2). Then it recursively marks the nodes that have likely produced erroneous outputs starting from the final erroneous nodes (lines 3 and 4). Finally, it checks all the erroneous nodes for the condition in the fourth observation i.e., being erroneous with no erroneous predecessor (lines 5 to 9). If the condition is satisfied, it increments the counters for the resources used by the erroneous nodes by 1.

Algorithm 1: Heurisitc 1

```
   input: resources
   Algorithm heuristic1
1    foreach node n of final erroneous nodes do
2      Sn1 = Sn2 = φ // Initializing sets
3      foreach node k of n.ancestors do
4        if k.isLastCorrect() then
5          Sn1.add(getNodesBetween(n , k))
       end
6      foreach R of resources do
7        if R is used in Sn1 then
8          counters[R]++;
       end
9      foreach node k of nodes in backward slice of
       n do
10       if not(k.hasFinalCorrectDescendent) then
11         Sn2.add(k)
       end
12     foreach R of resources do
13       if R is used in Sn2 then
14         if n.hasFaultyAncestor() then
15           counters[R] += 0.5
16         else
17           counters[R] += 1
       end
     end
```

Algorithm 2: Heurisitc 2

```
   Procedure markCorrect(node n)
     foreach node p of the predecessors of n do
       ec ← p.getErroneousChildrenCount()
       if not (p.isErroneous() OR ec ≥ 2) then
         p.correct ← True
         markCorrect(p)
       end
     end
   Procedure markErroneous(node n)
     foreach node p of the predecessors of n do
       cp ← p.getNonCorrectPredecessorsCount()
       cc ← p.getCorrectChildrenCount()
       if cp == 1 AND cc ≤ 1 then
         p.erroneous ← True
         markErroneous(p)
       end
     end
   Algorithm heuristic2
1    foreach node n of the final correct nodes do
2      markCorrect(n)
     end
3    foreach node n of the final erroneous nodes do
4      markErroneous(n)
     end
5    foreach node n of the erroneous nodes do
6      cond1 ← (n.erroneousParentsCount == 0)
7      cond2 ← (n.correctParentsCount ≥ 1)
8      if cond1 AND cond2 then
9        Increment Counters of resources used in n;
     end
```

After both heuristics are applied, the counter values computed by the heuristics are averaged to obtain the final counter values. The diagnosis algorithm identifies the top N_{deconf} resources with the highest counter values as candidates of the faulty resource, where N_{deconf} is a fixed value. These are the processor resources that are disabled to fix the intermittent fault after diagnosis. Thus N_{deconf} represents a trade-off between diagnosis accuracy and granularity. We study this trade-off in Section VI-B1.

In general, we disable all the N_{deconf} resources identified by the diagnosis algorithm, with one exception. Because the number of functional units in a processor is typically low, we never disable more than one functional unit. This means that if the number of functional units among the resources with N_{deconf} highest final counter values is more than one, only the unit with the highest counter value is disabled.

Example: Due to space constraints, we only demonstrate the application of the first heuristic to the augmented DDG in Figure 5. Heuristic 1 starts from erroneous nodes (nodes 2, 6, 7 & 8). None of the erroneous nodes in this DDG have a final correct ancestor and therefore $S_{21} = S_{61} = S_{71} = S_{81} = \phi$. The backward slice for each of the erroneous nodes are collected by the algorithm ($S_{22} = \{2, 1\}$, $S_{82} = \{8, 5\}$, $S_{72} = \{7, 5\}$, $S_{62} = \{6, 5\}$). The counters of resources in these sets are incremented by 1 as they have each participated in creating an erroneous value.

These nodes might *also* have participated in creating a final correct value. If so, they are pruned from the backward slice before their counters are incremented (Line 10). However, none of the nodes in the backward slices of the erroneous nodes in Figure 5 have final correct nodes as their children. Therefore, no *pruning* occurs in the example.

We can see that node 5 which has used the faulty resource *fu-1*, appears in the backward slices of three erroneous nodes (6, 7 & 8). This means that the counter related to *fu-1* is incremented 3 times. Meanwhile, *fu-1* is also used by the node 1 in the backward slice of erroneous node 2 (based on Table I), and hence its counter value is again incremented by 1. The final counter values are shown in Table III. As seen from the table, the faulty resource *fu-1* is the resource with the highest counter value of 4.

Resource	Value	Resource	Value
fu-1	4	fu-5	2
rob-84	3	rob-85	1
ifq-1	3	lsq-2	1
rs-7	3	...	1

Table III: Counter values after applying heuristic 1 to DDG in Figure 5

Fault Recurrence: The above discussion considers a single occurrence of an intermittent fault. However, by their very definition, intermittent faults will recur, thus giving us an opportunity to diagnose them again. The above diagnosis process is repeated after every failure resulting from an intermittent fault, and each iteration of the process yields a different counter value set. The final counter values are averaged across multiple iterations, thus boosting the diagnosis accuracy, and smoothing the effect of inaccuracies.

VI. Evaluation

We answer the following research questions to evaluate our diagnosis technique:

1) **RQ 1**: What is the diagnosis accuracy or the probability that the technique correctly finds the faulty resource?
2) **RQ 2**: What is the performance overhead of repairing the processor after finding the faulty resource?
3) **RQ 3**: How much online performance, power and area overhead is incurred because of SCRIBE?
4) **RQ 4**: What is the offline performance overhead of SIED (Replay + DDG Construction and analysis)?

In this section, we present the experimental setup and the results of our evaluations.

A. Methodology

SCRIBE: We implemented SCRIBE in *sim-mase*, a cycle-accurate micro-architectural simulator, which is a part of the SimpleScalar family of simulators [23]. We based our implementation on the SimpleScalar Alpha-Linux, developed as part of the XpScalar framework [24].

Configurations: To understand the overhead of our diagnosis mechanism across different processor families, we use three different configurations (Narrow, Medium and Wide pipelines) for our experiments. These respectively represent processors in the embedded, desktop and server domains, and have been used in prior work on instruction-level duplication [25]. Table IV lists the common configurations between the simulated processors and Table V shows the configurations that vary across processor families.

Parameter	Value
Level 1 Data Cache	32K, 4-way, LRU, 1-cycle latency
Level 1 Instruction Cache	32K, 4-way, LRU, 1-cycle latency
Level 2 combined data & instruction cache	512K, 4-way, LRU, 8-cycle latency
Branch Predictor	Bi-modal, 2-level
Instruction TLB	64K, 4-way, LRU
Data TLB	128K, 4-way, LRU
Memory Access Latency	200 CPU Cycles

Table IV: Common machine configurations

We choose the RUI length based on the type of the processor (recall from Section IV that *RUI Length* $\propto lg($*Total number of resources*$)$). We choose the LogSQ and Logging Buffer to be 32 and 64 entries respectively, as our experiments indicate that increasing the sizes of these resources beyond 32 and 64 makes no significant improvement on performance. More details may be found in our earlier paper [21].

Benchmarks: We use eight benchmarks from the SPEC 2006 integer and floating-point benchmarks set. We chose these benchmarks as they were compatible with our infrastructure. We did not cherry-pick them based on the results.

Fault Injector: We extended *sim-mase* to build a detailed micro-architecture level fault injector. For each injection, the program is fast-forwarded 20 million instructions to remove initialization effects. Then a single intermittent fault burst is injected into one of the following: **i)** Reorder Buffer entries, **ii)** Instruction Fetch Queue entries, **iii)** Reservation

Topic	Parameter	Machine Width		
		Nar.	*Med.*	*Wide*
Pipeline Width	Fetch	2	4	8
	Decode	2	4	8
	Issue	2	4	8
	Commit	2	4	8
Array Sizes	ROB Size	64	128	256
	LSQ Size	32	32	32
Number of Functional Units	Integer Adder	2	4	8
	Integer Multiplier	1	1	1
	FP Adder	1	1	2
	FP Multiplier	1	1	1

Table V: Different machine configurations

Station entries, **iv)** Load/Store Queue entries **v)** functional unit outputs. The starting cycle of the fault burst is uniformly distributed over the total number of cycles executed by the program. The number of cycles for which the fault persists (fault duration) is also uniformly distributed over the interval $[5, 2000]$, as voltage and temperature fluctuations last for around 5 to several thousands of cycles ([26], [27]).

After injecting the fault burst, the benchmark is executed and monitored for 1 million instructions to see if it crashes. We consider only faults that lead to crashes for diagnosis. This is because we do not assume the presence of error detectors in the program that can detect an error and halt it. To simulate a recurrent intermittent fault, we re-execute a benchmark up to 50 times while keeping the injection *location* unchanged. Note however that the starting cycle and fault duration are randomly chosen in each run. We report the results for scenarios in which 10 or more of the fault injections into a location led to crashes (out of 50 injections).

Diagnosis: SIED is implemented using Python scripts and starts whenever a benchmark crashes as a result of fault injection. We extract the traces required to build the program's DDG by modifying the MASE simulator. However, these traces would be extracted by a virtual machine or a dynamic binary instrumentation tool in a real implementation of SIED (as explained in Section V-A). SIED also relies upon deterministic replay mechanisms (as explained in Section III-C) for diagnosis. We have extended *sim-mase* to enable deterministic replay. Again, this would be implemented by a deterministic replay technique in a real implementation of SIED. We conducted the simulations and diagnosis experiments on an Intel Core i7 1.6GHz system with 8MB of cache.

Deconfiguration Overhead: The deconfiguration overhead is measured as the processor's slow-down after disabling the candidate locations of the faulty resource suggested by our diagnosis approach. We assume that the precise subset of resources suggested by our technique can be deconfigured. We used the medium width processor configuration from Table V for measuring the overhead after deconfiguration.

SCRIBE Performance and Power Overhead: The performance overhead is measured as the percentage of extra cycles taken by the processor to run the benchmark programs when SCRIBE is enabled. For measuring the overhead, we execute each benchmark for 10^9 instructions in the MASE simulator [1]. We also implemented SCRIBE in the Wattch simulator [28] to evaluate its power overhead. The metric by which the power overhead of SCRIBE is evaluated is the average total power per instruction. We used the CC3 power evaluation policy in Wattch as it also takes into account the fraction of power consumed when a unit is not used [28].

B. Results

1) Diagnosis Accuracy (RQ 1): Figure 6 shows the accuracy of our diagnosis approach for faults occurring in different units of the medium-width processor. We find that the average accuracy is 84% across all units. *To put this in perspective, our diagnosis approach identifies 5 resources out of more than 250 resources in the processor as faulty, and the actual faulty resource is among these 5 resources, 84% of the time (later, we explain why we chose 5).*

The diagnosis accuracy depends on the unit in which the fault occurs, and ranges from 71% for IFQ to 95% for LSQ. The reason for IFQ having low accuracy is that faults in the IFQ cause the program to crash within a short interval of time (i.e., they have shorter crash distances). Short crash distances lead to lower accuracy, which is counter-intuitive as one expects longer crash distances to cause loss in the fault information and hence have lower accuracy. However, our DDG analysis algorithm explained in Section V-B uses backtracking the paths leading to final erroneous data. The more the number of these paths, the easier it is for our algorithm to distinguish the faulty resource from other resources, and hence higher the accuracy. Shorter crash distances mean fewer paths, and hence lower accuracy.

The main source of diagnosis inaccuracies is that SIED has only knowledge about *final* data (correctness of memory and register values at the *failure point*). The DDG analysis heuristics in Section V-B use backtracking from final erroneous data to speculate on the correctness of the data before the failure point. However, non-faulty resources are also used in the paths leading to final erroneous data, and can be incorrectly diagnosed as faulty by our technique.

One way to improve the diagnosis accuracy is to record the output of every instruction, thus eliminating the need for speculation on the correctness of the data before the failure point. However, storing the output of every instruction imposes prohibitive performance overhead. Figure 7 shows the performance overhead of storing the destination register of every instruction, for 32-bit instructions and 64-bit instructions, for three SPEC 2006 programs. The overhead for storing 0 extra bits corresponds to that of storing only the resource usage bits, as done by our technique (explained in Section IV). As seen from the Figure 7, the overheads for storing the results of 32 and 64 bit instructions are respectively 2X and 3X that of the overhead of only storing the resource usage information. Therefore, we chose not to record the output of every instruction for diagnosis.

As explained in Section V-B, SIED uses information from multiple occurrences of the intermittent fault to enhance the diagnosis accuracy. Let RN denote the number of recurrences of the failure, after which the diagnosis is performed.

[1]We do not use Simpoints due to incompatibilities between the benchmark format for the simulator and the format required by Simpoints.

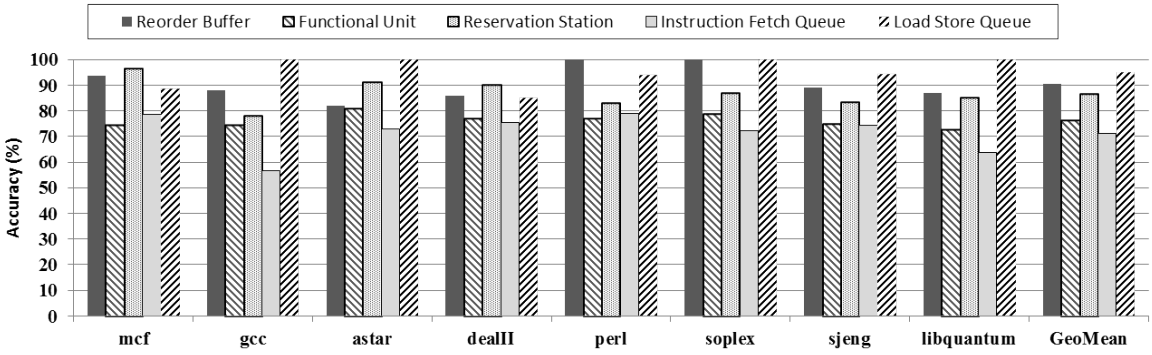


Figure 6: Accuracy Results for applying the heuristics ($RN = 4$ and $N_{deconf} = 5$)

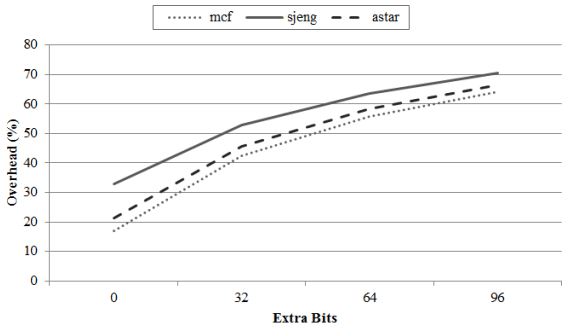


Figure 7: Effect of sending the destination register values of every instruction on performance overhead (0 bits corresponds to only sending the RUI as in our technique)

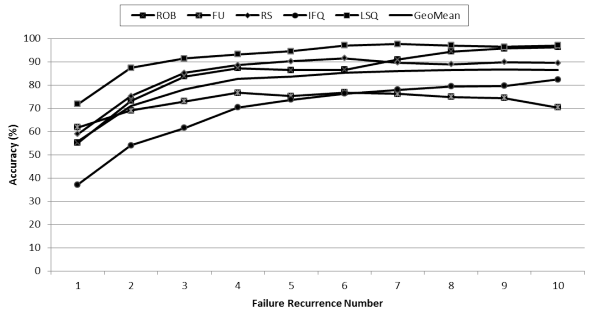


Figure 8: Average accuracy across benchmarks with respect to the number of failures ($N_{deconf} = 5$)

There is a trade-off among diagnosis accuracy and the failure recurrence number (RN) for performing diagnosis. This means that diagnosis can be performed earlier at the expense of less accuracy or be postponed to receive more information from the subsequent failures and hence achieve higher accuracy, which in turn decreases the probability of the fault recurring after deconfiguration (and hence has lower overheads). Figure 8 shows how changing the RN value can affect the accuracy of diagnosis. We choose $RN = 4$ to perform diagnosis (Figure 6), as beyond this point, there is only a marginal increase in diagnosis accuracy with increase in RN.

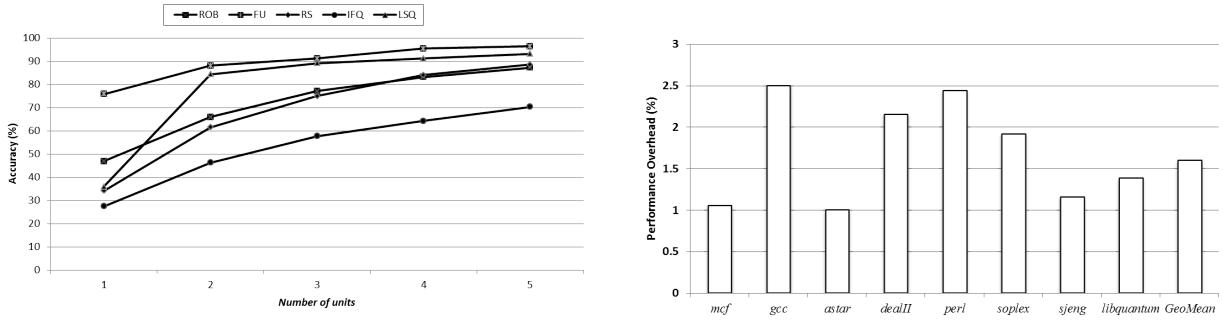
2) Deconfiguration overhead (RQ 2): As mentioned in section V-B, N_{deconf} is the number of resources suggested by SIED as most likely to be faulty. *Diagnosis accuracy* is defined as the probability of the actual faulty resource being among the resources suggested by SIED. For the accuracies reported in Figure 6, N_{deconf} is chosen to be 5.

The processor is deconfigured after diagnosis by disabling these N_{deconf} resources. Although increasing N_{deconf} increases the likelihood of the processor being fixed after deconfiguration, it also makes the granularity of diagnosis more coarse-grained. In other words, by increasing N_{deconf}, deconfiguration disables more non-faulty resources along with the actual faulty resource. This results in performance loss after deconfiguration.

Figure 9a shows the accuracy of diagnosis as N_{deconf} varies from 1 to 5. As expected, increasing N_{deconf} increases the accuracy of diagnosis to 84% for $N_{deconf} = 5$. Figure 9b shows the average slowdown by disabling $N_{deconf} = 5$ resources suggested by our technique. As can be seen in the figure, the slowdown varies from 1% to 2.5%, with an average of 1.6%. This shows that disabling $N_{deconf} = 5$ resources only incurs a modest performance overhead after reconfiguration, and hence we choose this value.

3) SCRIBE Performance, Power and Area Overhead (RQ 3): Figure 10 shows the performance overhead incurred by SCRIBE across three processor configurations, narrow, medium and wide, described in Section VI-A. The *geometric mean* of the overheads across all configurations is 14.7%. In all but one case (except *soplex*), the *wide* configuration ($GeoMean = 23.21\%$) incurs higher overhead than the *medium* ($GeoMean = 11.88\%$) and *narrow* ($GeoMean = 11.53\%$) configurations. The *Medium* and *narrow* configurations are comparable in terms of overhead. The *wide* processor has high overhead as it is able to utilize the resources better, thus leaving fewer free slots to be used by SCRIBE for sending logging stores to memory.

As far as power is concerned, SCRIBE has 9.3% power overhead on average. This includes both active power and idle power. Figure 11 shows the breakdown of the power



(a) Accuracy with respect to N_{deconf} (RN = 4)

(b) Performance overhead after deconfiguration ($N_{deconf} = 5$)

Figure 9: The reported values are averages of values for benchmarks mentioned in Section VI-A

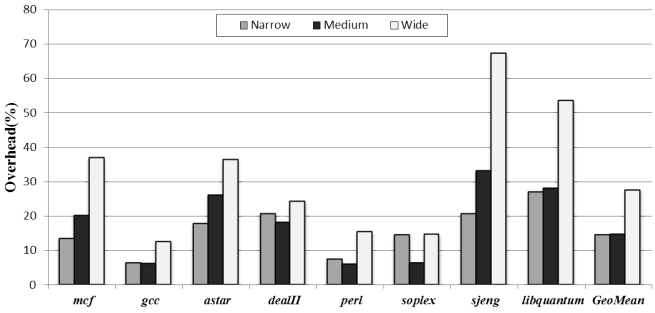


Figure 10: The performance overhead of SCRIBE applied to three configurations: *Narrow*, *Medium* and *Wide*

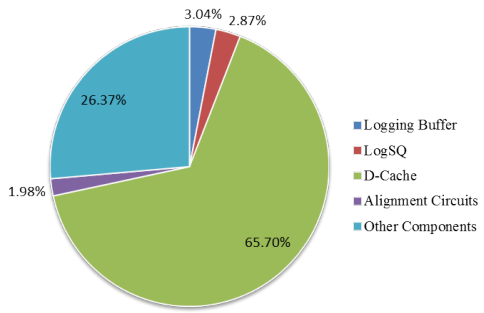


Figure 11: The breakdown of power consumption of SCRIBE

consumption overhead. As seen in the figure, only 7.9% of the extra power is used by the components of SCRIBE. The rest of the power overhead is due to the extra accesses to the D-Cache and the extra cycles due to SCRIBE (indicated in the figure as *Other Components*).

We have not synthesized SCRIBE on hardware, and hence cannot measure its area overhead. However, we can estimate the area overheads from other techniques that have been synthesized. For example, a comparable technique, IFRA, which add 50 Kbytes of storage to a chip, has an area overhead of 2% [11]. SCRIBE adds less than 2 KBytes of distributed on chip storage (estimated from the number of bits added by each component). Therefore, we believe the area overhead of SCRIBE will be much less than 2%.

4) SIED Offline Performance Overhead (RQ 4): This overhead consists of: **i)** Replay time **ii)** DDG construction and analysis time. The average replay time depends on the program and whether it is replayed from a checkpoint or from the beginning. We do not consider this time as it depends on the checkpointting interval. The DDG construction and analysis time took 2 seconds on average, for our benchmarks.

VII. Related Work

Bower et al. [5] propose a hardware-only diagnosis mechanism by modifying the processor pipeline to track the resources used by an instruction (similar to SCRIBE), and finding the faulty resources based on resource counters. However, their scheme relies on the presence of a fine-grained checker (e.g., DIVA [8]) to detect errors before an instruction commits. This limits its applicability to processors that are specifically designed with such fine-grained checkers.

Li et al. [9] use a combination of hardware and software to diagnose permanent errors. Similar to our approach, theirs is also a hybrid technique that splits the diagnosis between hardware and software. However, they rely on the determinism of the fault, as they replay a failed program execution (due to a permanent fault) from a checkpoint and gather its micro-architectural resource usage information during the replay. Unfortunately, this technique would not work for intermittent faults that are non-deterministic, as the fault may not show up during the replay.

IFRA [11], is a post-silicon bug localization method, which records the footprint of every instruction as it is executed in the processor. IFRA is similar to SCRIBE in how it records the information. However, SCRIBE differs from IFRA in two ways. First, IFRA records the instruction information within the processor, and this information is scanned out after the failure, after the processor is stopped. On the other hand, SCRIBE writes the gathered information to memory during regular operation. Second, IFRA required

the presence of hardware-based fault detectors to limit the error propagation. In contrast, SCRIBE does not require any additional detectors in the hardware or software.

DeOrio et al. [29] introduce a hybrid hardware-software scheme for post-silicon debugging mechanism, in which the hardware logs the signal activities during post-silicon validation, and the software uses anomaly detection on the logged signals to identify a set of candidate root-cause signals for a bug. Because their focus is on post-silicon debugging, they do not present the performance overhead of their technique, and hence it is not possible for us to compare their performance overheads with ours.

Carratero et al. [10] performs integrated hardware-software diagnosis for faults in the Load-Store Unit (LSU). Our work is similar to theirs in some respects. However, our approach covers faults in the entire pipeline, and not only the Load Store Unit. Further, their goal is to diagnose design faults during post-silicon validation, while ours is to diagnose intermittent faults during regular operation.

There has been considerable work on online testing for fault diagnosis. For example, Constantinides et al. [30] propose a periodic mechanism to run directed tests on the hardware using a dedicated set of instructions. However, this technique may find errors that do not affect the application, which in turn may initiate unnecessary recovery or repair actions, thus resulting in high overheads. To mitigate this problem, Pellegrini and Bertacco. [31] propose a hybrid hardware-software solution that monitors the hardware resource usage in the application, and tests only the resources that are used by the application. While this is useful, all testing-based methods require that the fault appears during at least one of the testing phases, which may not hold for intermittent faults.

VIII. Conclusion

In this paper, we proposed a hardware/software integrated scheme for diagnosing intermittent faults in processors. Our scheme consists of SCRIBE, the hardware layer, which enables fine-grained software layer diagnosis, and SIED, the software layer which uses the information provided by SCRIBE after a failure to diagnose the intermittent fault. We found that using SCRIBE and SIED, the faulty resource can be correctly diagnosed in 84% of the cases on average. Our scheme incurs about 12% performance overhead, and about 9% power consumption overhead (for a desktop class processor). The performance loss after disabling the resources suggested by our technique is 1.6% on average.

Acknowledgment

We thank the anonymous reviewers of DSN'14 and SELSE'13 for their comments that helped improve the paper. This work was supported in part by a Discovery grant and an Engage Grant, from the Natural Science and Engineering Research Council (NSERC), Canada, and a research gift from Lockheed Martin Corporation. We thank the Institute of Computing, Information and Cognitive Systems (ICICS) at the University of British Columbia for travel support.

References

[1] S. Borkar, "Microarchitecture and design challenges for gigascale integration," in *Keynote Speech, 37th International Symposium on Microarchitecture*, ser. MICRO, 2004.
[2] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.
[3] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs," ser. EuroSys, 2011, pp. 343–356.
[4] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Adapting to intermittent faults in multicore systems," ser. ASPLOS, 2008, pp. 255–264.
[5] F. A. Bower, D. J. Sorin, and S. Ozev, "A mechanism for online diagnosis of hard faults in microprocessors," ser. MICRO, 2005, pp. 197–208.
[6] S. Gupta, S. Feng, A. Ansari, and S. Mahlke, "StageNet: A reconfigurable fabric for constructing dependable CMPs," *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 5–19, Jan 2011.
[7] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, "Intermittent hardware errors recovery: Modeling and evaluation," ser. QEST, 2012, pp. 220–229.
[8] T. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," ser. MICRO, 1999, pp. 196–207.
[9] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou, "Trace-based microarchitecture-level diagnosis of permanent hardware faults," ser. DSN, 2008, pp. 22–31.
[10] J. Carretero, X. Vera, J. Abella, T. Ramirez, M. Monchiero, and A. Gonzalez, "Hardware/software-based diagnosis of load-store queues using expandable activity logs," ser. HPCA, 2011, pp. 321–331.
[11] S.-B. Park and S. Mitra, "IFRA: Instruction footprint recording and analysis for post-silicon bug localization in processors," ser. DAC, 2008, pp. 373–378.
[12] J. W. McPherson, "Reliability challenges for 45nm and beyond," ser. DAC, 2006, pp. 176–181.
[13] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," ser. DAC, 2003, pp. 338–342.
[14] C. Constantinescu, "Intermittent faults and effects on reliability of integrated circuits," ser. RAMS, 2008, pp. 370–374.
[15] C. Weaver and T. Austin, "A fault tolerant approach to microprocessor design," ser. DSN, 2001, pp. 411–420.
[16] H. Agrawal and J. R. Horgan, "Dynamic program slicing," ser. PLDI, 1990, pp. 246–256.
[17] 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*, vol. 25, no. 6, pp. 30–39, 2005.
[18] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," ser. VEE, 2008, pp. 121–130.
[19] M. Xu, R. Bodik, and M. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," ser. ISCA, 2003, pp. 122–133.
[20] A. Bondavalli, S. Chiaradonna, F. di Giandomenico, and F. Grandoni, "Threshold-based mechanisms to discriminate transient from intermittent faults," *IEEE Transactions on Computers*, vol. 49, pp. 230–245, Mar 2000.
[21] M. Dadashi, L. Rashid, and K. Pattabiraman, "SCRIBE: A hardware infrastructure enabling fine-grained software layer diagnosis," *Silicon Errors in Logic, System Effects (SELSE)*, 2013.
[22] C.-K. 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," ser. PLDI, 2005, pp. 190–200.
[23] E. Larson, S. Chatterjee, and T. Austin, "MASE: a novel infrastructure for detailed microarchitectural modeling," ser. ISPASS, 2001, pp. 1–9.
[24] N. Choudhary, S. Wadhavkar, T. Shah, H. Mayukh, J. Gandhi, B. Dwiel, S. Navada, H. Najaf-abadi, and E. Rotenberg, "FabScalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template," ser. ISCA, 2011, pp. 11–22.
[25] A. Timor, A. Mendelson, Y. Birk, and N. Suri, "Using underutilized CPU resources to enhance its reliability," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 1, pp. 94–109, Jan 2010.
[26] R. Joseph, D. Brooks, and M. Martonosi, "Control techniques to eliminate voltage emergencies in high performance processors," ser. HPCA, 2003, pp. 79–90.
[27] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, "Temperature-aware microarchitecture: Modeling and implementation," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 1, pp. 94–125, Mar 2004.
[28] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," ser. ISCA, 2000, pp. 83–94.
[29] A. DeOrio, Q. Li, M. Burgess, and V. Bertacco, "Machine learning-based anomaly detection for post-silicon bug diagnosis," ser. DATE, 2013, pp. 491–496.
[30] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "Software-based online detection of hardware defects: Mechanisms, architectural support, and evaluation," ser. MICRO, 2007, pp. 97–108.
[31] A. Pellegrini and V. Bertacco, "Application-aware diagnosis of runtime hardware faults," ser. ICCAD, 2010, pp. 487–492.