Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults

Man-Lap Li, Pradeep Ramachandran, Swarup K. Sahoo, Sarita V. Adve, Vikram S. Adve, Yuanyuan Zhou
Department of Computer Science
University of Illinois at Urbana-Champaign
swat@cs.uiuc.edu

Abstract

As devices continue to scale, future shipped hardware will likely fail due to in-the-field hardware faults. As traditional redundancy-based hardware reliability solutions that tackle these faults will be too expensive to be broadly deployable, recent research has focused on low-overhead reliability solutions. One approach is to employ low-overhead ("always-on") detection techniques that catch high-level symptoms and pay a higher overhead for (rarely invoked) diagnosis.

This paper presents trace-based fault diagnosis, a diagnosis strategy that identifies permanent faults in microarchitectural units by analyzing the faulty core's instruction trace. Once a fault is detected, the faulty core is rolled back and re-executes from a previous checkpoint, generating a faulty instruction trace and recording the microarchitecture-level resource usage. A diagnosis process on another fault-free core then generates a fault-free trace which it compares with the faulty trace to identify the faulty unit. Our result shows that this approach successfully diagnoses 98% of the faults studied and is a highly robust and flexible way for diagnosing permanent faults.

1. Introduction

As we continue to scale CMOS, hardware reliability threatens to be a major challenge to reaping the benefits of Moore's law. Permanent faults due to phenomena such as wear-out and infant mortality are growing concerns for in-field processor reliability. The problem is pervasive across the broad computing market; therefore, there has been much emphasis on solutions that incur limited area, power, and performance overheads. Traditional manufacturing, device, and circuit level solutions often make worst-case, overly conservative assumptions. An alternative is to use an aggressive design that risks faults but relies on microarchitecture and higher system levels to detect these faults on-line.

While traditional high-level detection involves core-level redundancy, more recent techniques propose very low-cost monitoring of high level symptoms of faults. For example, Wang and Patel consider traps and mispredictions of high confidence branches as symptoms of transient faults [19]. Racunas et al. dynamically predict the valid set of values that an instruction will produce, and consider a departure from this prediction as a symptom of a (transient) fault [14]. Dimitrov and Zhou monitor the variance between the two most recent results produced by two dynamic instructions of the same PC, and any large variance indicates a possible soft error [5]. We have recently explored various symptoms – fatal traps, hangs, high OS activity, and program based invariant violations – as the mechanisms for detecting permanent and transient faults in our SWAT system [6], [15]. Meixner et al. describe a set of high-level detection techniques for a simple processor [9]. Such high-level detection mechanisms can be very effective because they provide coverage for a wide range of fault sources and faulty components.

Unlike transient faults, permanent faults require diagnosis in addition to detection. To ensure continuous operation, it is important to diagnose the source of the permanent fault so it can be repaired or reconfigured; e.g., by disabling the faulty component (such as a faulty core, ALU, or entries in a buffer, queue, or cache), reducing the frequency of operation of the component, or using software to replace the faulty execution of a specific instruction.

While there has been significant recent work on high-level detection of in-field faults, there is relatively little work on diagnosing the source of a permanent fault detected in this way. The higher the level at which a fault is detected, the longer the latency between the actual fault activation and detection and the more difficult it is to diagnose its root cause for repair. Therefore, to reap the benefits of emerging low-cost high-level detection techniques, we need to develop effective diagnosis techniques. This paper concerns such a diagnosis framework.

When a fault is detected and identified to a particular core, a simple option is to decommission that core [10]. However, that can be wasteful, especially since modern pro-

This work is supported in part by the Gigascale Systems Research Center (funded under FCRP, an SRC program), the National Science Foundation under Grants NSF CCF 05-41383, CNS 07-20743, and NGS 04-06351, an OpenSPARC Center of Excellence at the University of Illinois at Urbana-Champaign supported by Sun Microsystems, and an equipment donation from AMD.

cessors offer opportunities to reconfigure around individual (failed) microarchitectural components. We therefore investigate a technique to diagnose the fault to the granularity of a microarchitectural component. Our technique, Trace Based Fault Diagnosis (TBFD), is based on the following observations.

- It is acceptable to incur high overhead for the diagnosis procedure since it is invoked only in the infrequent case when a fault is detected (in contrast, the detection mechanism needs to be low overhead since it must be on all the time).
- Many detection schemes today rely on a checkpoint/restart mechanism for recovery [1], [9], [14], [19]. Our diagnosis exploits this mechanism to roll the faulty core back to a pristine checkpoint. It replays from the checkpoint to generate a detailed record of the execution trace that activated the fault. This trace can now be used as a starting point for instruction-based (functional) diagnosis.
- We exploit the trend towards multicore systems by using a fault-free core to generate a fault-free trace against which the replay on the faulty core can be compared to reveal diagnostic information. Effectively, we cheaply synthesize Dual-Modular Redundancy (DMR) for diagnosis, in contrast to expensive always-on DMR traditionally used for detection.

Our overall scheme proceeds as follows. We assume a single-threaded program that is executing on a modern out-of-order superscalar processor. The presence of the fault in the core is detected through one of the many existing detection mechanisms [6], [9], [14], [19]. For this work, we use the low-cost detection methods in the SWAT system to detect faults within the processor core [6]. Detection of a permanent fault triggers the firmware-driven TBFD algorithm on a fault-free core. TBFD rolls the faulty core back to a pristine checkpoint and replays the execution on it, while recording detailed information such as microarchitectural resource usage for all instructions. TBFD also downloads the checkpoint from the faulty core to the fault-free core, and plays a "golden" execution on the fault-free core. It compares the traces from the fault-free and faulty cores and systematically analyzes any points of divergence to accurately diagnose the faulty microarchitectural structure.

We evaluate the effectiveness of TBFD using microarchitecture-level fault injection experiments in a full-system simulation of an out-of-order processor. We apply TBFD to all the faults that are detected using the SWAT symptom-based fault detectors [6]. TBFD correctly diagnosed 89% of the detected faults to a single non-array structure (e.g., one particular ALU) or a single entry in an array structure (e.g., a particular register in the register file). Of the remaining detected faults, TBFD is able to correctly diagnose almost all of them to the correct array structure (as opposed to a specific entry in the array). The array entry could be narrowed down using traditional BIST-like techniques. With the above faults included, the accuracy of TBFD is 98%. Overall, TBFD presents a flexible and robust method for in-field microarchitecture level fault diagnosis.

2. Related Work

Addressing in-field permanent faults at higher system levels is a relatively new area with little work on high-level on-line diagnosis. The most related prior work is by Bower et al. [2], proposed in the context of the DIVA architecture [1]. Their scheme associates a counter for each reconfigurable (repairable) microarchitectural resource. As instructions flow through the pipeline, it keeps track of the microarchitectural resources used (e.g., which ALU, etc.) in a bit vector which is carried along through the pipeline. When a mismatch between the main processor and the DIVA checker is detected, the counter corresponding to each resource touched by the mismatching instruction is incremented. Once a resource counter reaches a certain threshold value, it is declared faulty. Our scheme differs from Bower et. al.'s scheme in the following ways. First, we incur diagnosis related overhead only in the infrequent case when a fault is detected. Their scheme, however, contains always-on monitors that present overheads in power and performance even in the common fault-free operation. Second, although their method works well for faults on the data path, it is not well-suited to handle faults in structures that establish or rely on logical to physical name translations. For example, a fault in a register alias table (RAT) entry or physical register number in a reorder buffer entry (ROB) is not handled by their technique [2]. TBFD diagnoses the faulty microarchitectural structure even in these scenarios.

Periodic low-level (conventional) tests to detect faults at a finer granularity, and using the signature of the test output to diagnose the fault source have also been proposed [4], [16]. Our diagnosis framework does not require the overhead (in performance, power, or increased wear-out) of periodic testing. Our overhead is incurred only in the infrequent case when a fault is actually detected through very low-cost detection techniques (e.g., [6]). Further, the coverage of the test-based approaches is limited by the fault models used to generate the test vectors. In contrast, our "test vector" is the program for which the fault was detected and is therefore known to excite the fault. Nevertheless, for parts of the chip where our approach does not provide sufficient diagnosis, conventional test-based schemes may be used to complement the trace-based diagnosis scheme and to improve its accuracy.

Of late, there has been much interest in logic self-test. Specifically, recent projects have explored instruction-sequence based test (also referred to as functional or embedded self-test) [3], [12], [20] where the processor generates its own (pre-specified) set of instructions for

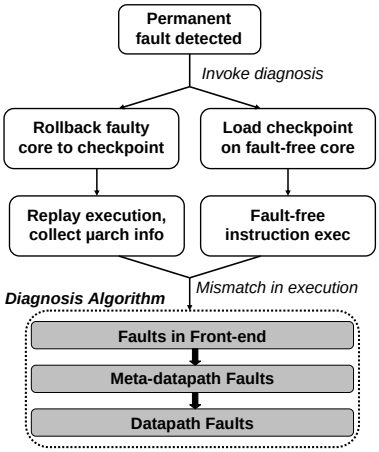


Figure 1. **TBFD overview.**

testing. The FRITS tool in particular has been extensively used for testing real x86 and Itanium processors [12]. In our diagnosis framework, such pre-specified instruction traces are not generated because the program and inputs for the execution of interest is already known. However, such mechanisms that generate pre-specified instruction sequences can aid in refining the resolution of diagnosis in the cases that the existing program inputs fail in providing sufficient information for accurate diagnosis. These form interesting future studies, but are beyond the scope of this paper.

3. Trace-Based Fault Diagnosis

This section describes the different components of our trace-based fault diagnosis algorithm (TBFD) and its implementation. On detection of a fault in a core, TBFD is invoked in the firmware of a fault-free core. TBFD must first determine whether it is a permanent fault (Section 3.1). It then constructs a detailed trace of the faulty execution on the faulty core (Section 3.2). This is followed by the generation of a corresponding fault-free execution on the fault-free core, and a resulting *test trace* that incorporates information from both the faulty and fault-free executions (Section 3.3). TBFD then analyses the test trace to finally diagnose the faulty structure (Section 3.4). Section 3.5 provides details on the implementation for TBFD. Section 3.6 gives a possible alternative strategy for TBFD and compares it to the chosen strategy. Figure 1 shows an overview of TBFD.

3.1. Identifying Permanent Faults

When a symptom is detected in a core, the diagnosis firmware rolls the core back to the previous checkpoint and replays the execution. If the symptom does not recur, it is diagnosed as a transient. If the symptom occurs again, the diagnosis firmware loads the checkpoint onto another fault-free core and replays the execution. If the symptom does not recur in the other core, a permanent fault is diagnosed in the original symptom-causing core. [1]

3.2. Generating the Detailed Faulty Trace

To generate the faulty trace, TBFD rolls the faulty core back to the previous checkpoint and replays the execution for a predefined number of instructions. It records a trace of the execution with the following information for each retired instruction:
Decode: Decoded opcode, logical source and destination register identifiers.
Data values: Values read from the source registers and written into the destination register. For loads, stores, and branches, it also records the virtual address.
Microarchitectural resources used by the instruction: For example, the source and destination physical register numbers, the specific functional unit used, etc. The specific information recorded depends on the reconfigurable units supported in the processor and consequent granularity of diagnosis required.

Section 3.5 describes hardware support for obtaining and recording the above information.

3.3. Fault-Free Execution and Test Trace

Next, the fault-free core is loaded with the checkpoint of the faulty core and the execution is replayed. For each instruction in this execution, the TBFD firmware compares the decode and value fields from the corresponding instruction in the faulty trace. Any mismatches in these fields cause the firmware to mark the corresponding instruction in the faulty trace as *mismatched* and record the cause for the mismatch. Additionally, the firmware synchronizes (corrupts) the fault-free core's state to that of the faulty core. This allows the fault-free core to continue executing a path similar to the faulty core until the next activation of the fault.

It is also possible that the corresponding instruction on the faulty core was hung at the head of the reorder buffer and never retired because it waits for its source operand(s) indefinitely. The firmware marks such an instruction in the faulty trace as *hung*. We assume hooks are available to extract information of the hung instruction even though it does not retire. When a hung instruction is encountered, the analysis algorithm diagnoses the source of the fault by examining the test trace (Section 3.4). If the algorithm does not terminate after the analysis, both the faulty core and the fault-free core are rolled back to generate a new test trace for further analysis.

We refer to mismatched and hung instructions collectively as *misbehaved instructions*. We refer to the faulty trace enhanced with the information about misbehaved instructions as the *test trace.*

1. If the symptom occurs in both cores, a software fault is diagnosed and handed over to the software layer.

3.4. Analysis of the Test Trace

The heart of the TBFD algorithm is the analysis of the generated test trace to to diagnose the fault. This analysis can be performed after completing building of the test trace. Alternatively, it may be periodically invoked after generating every N instructions of the test trace. The latter strategy may be more efficient if memory space to store the trace is at a premium. It also allows terminating test trace generation as soon as the diagnosis is able to uniquely identify the faulty structure.

TBFD divides the processor core into three different parts, on the basis of the information and analysis required to diagnose a fault in these parts:

1) *Front-End:* A fault in this part of the processor affects which instruction is executed, which operation is executed, and the logical source and destination registers accessed.
2) *Meta-Datapath:* Modern out-of-order processors use register renaming to translate logical register names to physical registers. Even if the front-end supplies the correct logical names, a fault in the translated name can result in erroneous computation. This type of fault is the largest source of complexity in TBFD – as we will show later, a corruption in the physical register name may not be caught by analyzing only the mismatched instructions. We use the term meta-datapath to refer to the parts of the core where a fault can corrupt the physical register name.
3) *Datapath:* This is the conventional data path, including the functional units, buses, and data residing in the physical register files.

In our work, we inject faults in the following structures as representatives of each of the above categories (see Table 2): *Front-end*: Instruction decoders. *Meta-datapath:* Register alias table (RAT) entries; source and destination (physical) register identifier fields in the reorder buffer (ROB).[2] *Datapath:* ALU, address generation unit, register data bus, and integer physical registers.

The analysis described below assumes faults in only the above structures, but can be extended to others as well.[3]

The analysis algorithm proceeds by using misbehaved instructions in the test trace as the starting point of the diagnosis. On encountering a misbehaved instruction in the trace, the algorithm systematically analyzes the misbehavior and determines if it can conclusively identify a fault in a unique structure. If so, it successfully terminates; otherwise, it updates counters corresponding to the microarchitectural resources used by the misbehaved instruction in the test trace. It then moves on to analyzing the next misbehaved instruction. If at any stage, one of the resource counters reaches a value higher than any other counters, the algorithm declares that resource as faulty and terminates. If the end of the trace is reached, then the algorithm identifies the resources with the highest value counters as suspected faulty units – in this case, it is not able to uniquely identify a faulty resource.

Next we describe how TBFD systematically analyzes the misbehaved instructions to track down faults to the three targeted areas in the processor.

3.4.1. Faults in Front-End. If the misbehaved instruction is a mismatched instruction (i.e., not hung), TBFD first suspects a front-end fault. (As will be seen later, a hung instruction can only arise from a meta-datapath fault.) For this, it simply needs to check if the test trace indicates that the mismatch occurred in the decode information – such a mismatch indicates that the instruction word was corrupted at the front-end. For example, when the faulty instruction uses r_1 as source operand but the fault-free instruction uses r_3 as source operand, a fault is suspected in the front-end. Consequently, counters of the front-end units used in the faulty execution are incremented. In this study, since only decoders are accounted for in the front-end, the first mismatch in the instruction word makes the decoder used by the mismatching instruction identified as the unique faulty unit and successfully terminates the algorithm.

3.4.2. Faults in Meta-Datapath. If either the misbehaved instruction was hung or if it was a mismatched instruction and no front-end fault was identified, then TBFD analyzes the misbehavior to check for meta-datapath faults.

This class of faults requires the most sophisticated analysis method. This is because, unlike the front-end and datapath, the first instruction that is affected by such a fault may not appear as a misbehaved instruction; i.e., it may not affect the fields in the faulty trace that are compared with the fault-free execution. Instead, it may silently corrupt processor architectural state, causing later unrelated instructions to misbehave and obscuring the real source of the fault.

For example, in Figure 2, I_a writes to r_3 which is mapped to physical register p_{23} and I_c reads from r_3. I_b writes to r_1 but is incorrectly mapped to p_{23} because of a meta-datapath fault (e.g., the register alias table had the wrong mapping). Thus, when I_b executes, r_3 is corrupted with the value of r_1; however, this is not indicated in any way in the information recorded for I_b in the test trace. Now when I_c retires, it sees the wrong value. This is caught when the faulty trace is compared with the fault-free execution and I_c is marked as a mismatched instruction. Now if TBFD were to blindly attribute this mismatch to the datapath structures used by I_c, the actual meta-datapath fault will never be identified.

In this study, TBFD focuses on meta-datapath faults in

2. In a real implementation, source register identifier fields would be in the issue queue; however, our simulator models them in the ROB and our algorithm uses the same terminology.

3. The algorithm assumes Intel Pentium 4 style register renaming with a distinct retirement register alias table or RRAT.

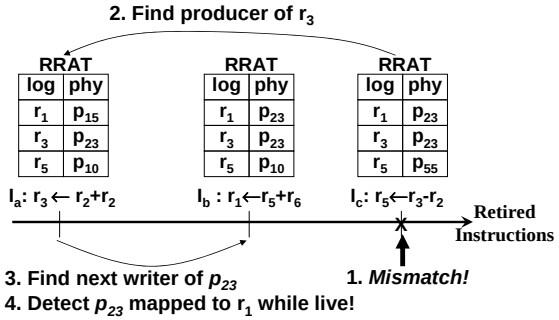


Figure 2. **An example scenario depicting how a physical register that is mapped to more than one logical register is identified by TBFD.**

the ROB and RAT entries. In particular, TBFD checks the integrity of the logical-physical register mappings of the misbehaved instruction based on the following two conditions of fault-free executions.

1) *A non-free physical register can be mapped to at most one logical register at any time.*
2) *If an instruction reads from physical register p_x that is mapped to logical register r_y, the last instruction that writes to logical register r_y (the producer) must have written to physical register p_x.*

If a fault occurs in the meta-datapath, one or both of the above conditions may not hold. The first condition above handles the case discussed in Figure 2, where instruction I_c is detected as a mismatching instruction (step 1). To check if condition 2 is violated, TBFD searches backward in the test-trace to check the integrity of the mappings of I_c's registers. Thus, it finds register r_3's producer, instruction I_a, that maps r_3 to physical register p_{23} (step 2). To verify that condition 1 holds, TBFD searches forward from I_a for the next writer to p_{23} and finds that I_b maps r_1 to p_{23} (step 3) while it is still mapped to r_3 (step 4) and condition 1 is violated. Consequently, TBFD increases the counters of the RAT entries for both r_1 and r_3 since it does not know where the fault is located. Nonetheless, with more misbehaved instructions, the faulty RAT entry can be identified.

Condition 2 is usually violated by a ROB fault. To check if condition 2 holds, TBFD goes backwards in the test-trace from the misbehaved instruction to the producing instruction and verifies its logical to physical register mappings. For example, a fault in the destination register number field causes instruction I_A to write to a different physical register than indicated in the RAT. Then, a dependent instruction I_B reads the mapping from the RAT and waits indefinitely for a physical register that will never be set ready by I_A. As a result, I_B becomes a hung instruction. TBFD then starts tracing from I_B to find that condition 2 is violated. As a result, TBFD increments the counter of the ROB entries of both I_A and I_B. With more misbehaved instructions, the faulty ROB entry can be uniquely identified.

However, even with techniques described above, RAT faults that are exercised by speculative instructions can be hard to diagnose down to the individual RAT entries. The scenario described below illustrates the difficulty. Consider that a logical register r_1 is mapped to a physical register p_1. Suppose an instruction *I* that writes to logical register r_2 enters the rename stage. Because of a fault in the RAT entry, r_2 gets mapped to the already live physical register p_1. Then, I executes, writes to p_1, and wipes out r_1's data. Later on, I is squashed as a result of an exception or a branch mis-prediction, causing p_1 to be freed and added to the free list (even though it is supposed to be live and mapped to r_1). Subsequently, when another logical register is mapped to p_1 and written by another instruction that retires and becomes architecturally visible, r_1 now shows a corruption in the architectural state as its value is now incorrect. However, since TBFD never looks at the intervening speculative instruction *I* (remember that TBFD only tracks retiring instructions), the faulty RAT entry is not correctly identified. Nevertheless, with more misbehaved instructions diagnosed, TBFD is able to identify the existence of RAT faults.

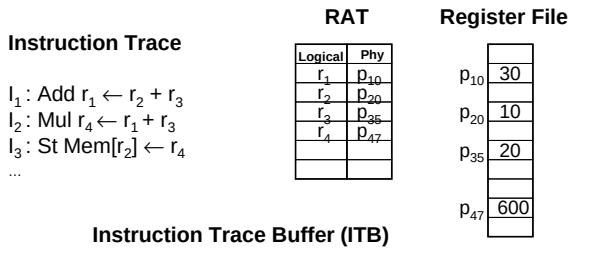
3.4.3. Faults in Datapath. After TBFD determines that a mismatched instruction is unlikely to have been caused by a fault in the front-end or the meta-datapath, a fault in the datapath is suspected. At this point, the microarchitectural structures (the functional unit, the result bus, and the destination physical register) on the datapath that are used by the misbehaved instruction are deemed potentially faulty. As a result, the counters of these structures are incremented. With more misbehaved instructions analyzed, the faulty module is likely to be the most frequently used with the highest counter value among all structures and thus can be identified.

3.5. Implementation

The TBFD algorithm is implemented in firmware. The detection of a fault on a core must result in an interrupt on another core (possibly through a protected channel) where the control transfers to the diagnosis firmware on that core. A single-core fault model implies that the latter core is fault-free; otherwise, the system must provide a protected, possibly simpler, fault-free core to invoke for diagnosis and recovery. (Analogous support is likely required for multicore systems that aim to provide continuous operation in the presence of a non-repairable fault in a core.)

Additionally, the system must support checkpoint generation for the faulty core and checkpoint migration to a fault-free core. Several techniques have been proposed for checkpointing for the purpose of recovery from hardware failures [13], [17], and can be used for TBFD as well. For example, the SafetyNet scheme [17] could be used, with the checkpointed state made accessible to firmware on other cores.

The most significant hardware support required for TBFD pertains to the generation of the test-trace. For



Instruction Trace Buffer (ITB)

	Decode information				*µarch resource info*				*Data values*			
	op	***s***	***t***	***d***	p_s	p_t	p_d	***Unit***	val_s	val_t	val_d	***Addr***
I_1	**Add**	r_2	r_3	r_1	p_{20}	p_{35}	p_{10}	alu_1	**10**	**20**	**30**	**-**
I_2	**Mul**	r_1	r_3	r_4	p_{10}	p_{35}	p_{47}	alu_2	**30**	**20**	**600**	**-**
I_3	**St**	r_2	r_4	**-**	p_{20}	p_{47}	**-**	stq_7	**-**	**-**	**600**	**0x10**

Figure 3. **An example Instruction Trace Buffer (ITB). For each instruction retired by the faulty core in diagnosis-mode, the ITB records information pertaining to 1) decoded instruction information, 2) some microarchitectural resources used by the instruction, and 3) the data values used by the instruction.**

this purpose, we propose to use an Instruction Trace Buffer or ITB, illustrated in Figure 3. Since diagnosis is not performance-critical, the ITB could be implemented entirely in memory or in cache. For better efficiency, we propose an on-chip hardware FIFO buffer that is periodically flushed to memory.

On the faulty core, the ITB is responsible for storing three types of information for each retired instruction: the decoded instruction information, the microarchitectural resources used by the retiring instruction, and the data values of the retiring instruction. The decoded information of each instruction includes the instruction opcode, the source operands, and the destination operands. The microarchitectural resources usage information refers to microarchitectural structures (e.g., decoder, functional units, source and destination physical registers, etc.) that were used by the retiring instruction. The data values of the retiring instruction corresponds to the source values, destination value, and the virtual address used in the case of a load, store, or branch. Figure 3 gives an example of an ITB for a small retirement trace from a faulty core.

Populating the fields of the ITB: Since the ITB is populated only in the rare event of a fault, we propose to populate the ITB with additional circuitry that taps into current microarchitectural structures for this information. An entry in the ITB is allocated once the instruction is decoded, with decode information from the decoder. When the instruction is allocated a ROB entry, and added to an issue queue, microarchitecture-level usage information (such as the physical registers used, ROB entry occupied, ALU used, etc.) can be populated. When the instruction writes its result, the data values corresponding to the instruction (destination register value and address) can be stored. If, however, the instruction is flushed, the corresponding entry from the ITB must be discarded as the trace accounts only for retiring instructions.

While the ITB and its upstream and downstream logic would incur area overhead, they are only activated during diagnosis after a rare event of a detection. During fault-free execution, these modules can be power-gated to reduce the power and performance overhead during normal operations. This is in contrast to previous methods of obtaining such information by adding bits that flow along with the instructions throughout the pipeline [2].

Diagnosis granularity and size of ITB: The granularity at which TBFD can diagnose a faulty microarchitectural unit is governed by the level of detail at which information is recorded in the ITB, which in-turn determines the size of the ITB. The fields to record in the ITB can be determined based on the level of repair supported by hardware. For example, if the hardware only supports replacing an entire array, as opposed to individual entries in the array, the ITB needs to only record the fact that this array was accessed, and not the specific entry in the array that was accessed. In our simulations, we assume that fine-grain reconfiguration is supported for the parts of the front-end, meta-datapath, and datapath which may contain faults (Section 3.4) and record their usage information in the ITB.

Test-trace generation and analysis: On the fault-free core, the firmware performs the golden execution from the faulty core's checkpoint, comparing instructions from the golden and faulty executions. On a misbehaved instruction, it needs to corrupt the golden state and enhance the faulty trace with bits to indicate the source of the misbehavior, thereby generating the test-trace. These bits are best implemented as extensions to the ITB. Since the golden execution is already emulated in software, it is unlikely to benefit from any acceleration due to hardware support of the ITB. Therefore, the additional bits above need not be implemented in the hardware FIFO for the ITB, and can simply be maintained in software. Finally, the analysis algorithm is invoked on the generated test trace – this algorithm works entirely in software.

3.6. Alternative Strategy for TBFD

The TBFD description above suggests that the fault-free core's state is synchronized to the faulty core's bad state when a mismatch occurs between the two executions. We also considered an alternative where the faulty core is synchronized to the fault-free core's good state when a mismatch is encountered. We refer to this alternative as the "patching" (versus corrupting) execution. A possible advantage of this alternative is that in diagnosis mode, the faulty core is made to go through the original program rather than potentially arbitrary code and data, while still communicating the impact of the fault on this code. We

implemented this method and did not find better diagnosis coverage than the corrupting method. We preferred the corrupting method since it is much easier to implement as follows.

In the corrupting version we chose, we did not have to execute the fault-free and faulty cores in synchrony. In fact, the entire faulty trace could be generated before the fault-free core started execution. The firmware on the fault-free core took care of corrupting the fault-free execution. In the patching version, this is not possible because the firmware cannot run on the faulty core. The faulty core must instead run roughly synchronized with the fault-free core. It must send the results of its instructions to the fault-free core and the fault-free core must send back any patches if needed. This is clearly much more complex and higher overhead than the corrupting version. Additionally, it requires the faulty core to patch the register file with data from the fault-free core while not knowing whether the path for overwriting the register file is fault-free.

It is interesting to note that the patching mode closely resembles the scheme proposed by Bower et al. where the DIVA checker is essentially the fault-free core that patches the architectural state of the faulty core [2]. While this is feasible in a tightly coupled scenario like DIVA, in a general multicore environment, it requires too *tight* lockstepping of two cores to be widely deployable.

4. Experimental Methodology

4.1. Simulation Environment

We use a full system simulation environment comprising the Wisconsin GEMS microarchitectural and memory timing simulators [7] in conjunction with the Virtutech Simics full system functional simulator [18]. Together, these simulators provide cycle-by-cycle microarchitecture-level timing simulation of a real workload (6 SpecInt2000 and 4 SpecFP2000) running on a real operating system (full Solaris-9 on SPARC V9 ISA) on a modern out-of-order superscalar processor and memory hierarchy (Table 1).

The GEMS + Simics infrastructure is based on the timing-first approach for simulation [8]. In this approach, the cycle-accurate GEMS timing simulator first executes an instruction. When this instruction is ready to retire, the functionally accurate Simics executes the same instruction. The resulting states are compared for coherence and in the case that they don't match (which may arise because GEMS does not implement a small subset of infrequently used instructions in the SPARC ISA), the timing simulator's state is updated with that from the functional simulator which is assumed to be accurate.

For our fault injections, we inject a single fault into the timing simulator's microarchitectural state and propagate it as the faulty values are read through the system. When a mismatch in the *architectural state* of the functional and the timing simulator is detected, the functional simulator (Simics) is corrupted if it is due to the injected fault. Otherwise, the value is read from Simics to GEMS, upholding the timing-first paradigm.

Base Processor Parameters	
Frequency	2.0GHz
Fetch/decode/execute/retire	4 per cycle
Functional units	2 Int add/mul, 1 Int div 2 Load, 2 Store, 1 Branch 2 FP add, 1 FP mult 1 FP div/Sqrt
Integer FU latencies	1 add, 4 mul, 24 div
FP FU latencies	4 default, 7 mul, 12 div
Reorder buffer size	128
Register file size	256 integer, 256 FP
Load-store queue	64 entries
Base Memory Hierarchy Parameters	
Data L1/Instruction L1	16KB each
L1 hit latency	1 cycle
L2 (Unified)	1MB
L2 hit/miss latency	6/80 cycles

Table 1. Parameters of the simulated processor.

μarch structure	**Fault location**
Instruction decoder	Input latch of one of the decoders
Integer ALU	Output latch of one of the int ALUs
Register bus	Bus on the write port to the reg file
Physical int reg file	A physical reg in the int reg file
Reorder Buffer (ROB)	Src/dest reg # of instr in ROB entry
Reg Alias Table (RAT)	Logical→ phy map of logical reg
Address gen unit (AGEN)	Virtual address generated by the unit

Table 2. Fault injection locations.

4.2. Faults Diagnosed

The focus of this study is to diagnose the permanent faults that are detected by the SWAT system. We injected 11,200 stuck-at and dominant-0 and dominant-1 bridging faults in various microarchitectural components (listed in Table 2) in 40 random points (in both time and space) during application execution. The injected faults are then simulated for 10M instructions in detailed timing simulation during which the low-cost software-symptom detectors in the SWAT system detect these faults. This methodology is identical to that in [6].

The detection techniques achieve a high coverage by detecting 95% of the non-masked faults, detecting approximately 8500 faults. These faults are subject to diagnosis using our TBFD algorithm, to identify the faulty microarchitectural component.

4.3. Implementation Assumptions

Emulating fault-free execution: We emulate the fault-free execution by exploiting the inherent dual execution mode prevalent in our simulation because of the timing-first simulation paradigm. When a fault is detected, the faulty execution is rolled back and replayed in the GEMS timing simulator, as it would in a real system. For the fault-free execution, we use the Simics functional simulator that runs in parallel with the timing simulation. Copying the state corrupted in the timing simulator due to the fault to

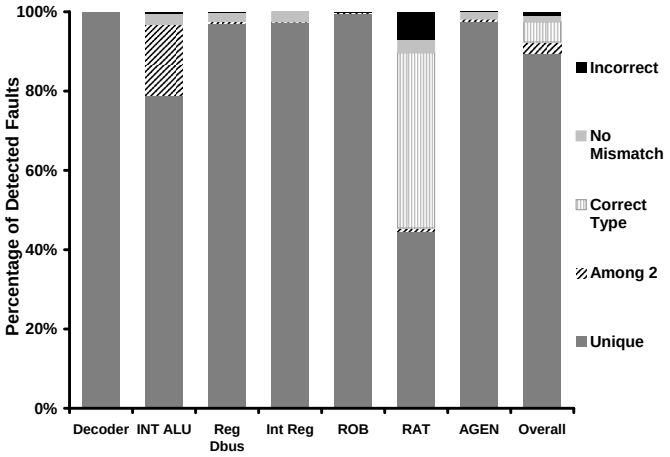


Figure 4. **Effectiveness of microarchitecture-level fault diagnosis. The figure shows the ability of the diagnosis algorithm to accurately diagnose detected faults. Overall, 98% of the detected faults are accurately diagnosed as either (1) the correct non-array structure or the correct entry within an array structure (the Unique stack); or (2) within one of two non-array structures or entries of array structures (Among 2); or (3) the correct array structure type but not the correct entry within the structure (Correct Type).**

the functional simulator corresponds to synchronizing the fault-free and faulty execution values. This process allows detection of misbehaved instructions as soon as they retire; therefore, the test-trace is also immediately generated and saved to a simulated ITB.

Checkpointing: In our simulations, fault-free checkpoints are recorded at the beginning of the execution, prior to fault injection. Rollback is implemented by reloading the register state, the TLB state, and rolling back the changes in the cache and memory state (similar to SafetyNet [17]).

Trace length: We run the faulty and the fault-free executions for up to 30 million instructions from the checkpoint. For efficiency, we invoke the TBFD analysis every 10,000 instructions collected in the ITB. If the algorithm finds the unique faulty structure, we terminate the simulation.

5. Results

Figure 4 presents the results indicating the effectiveness of the diagnosis for faults in different microarchitectural structures. In each bar, the *Unique* stack represents cases that the diagnosis process correctly and uniquely diagnoses the faulty non-array structure or the faulty entry within an array structure. The *Among 2* stack represents cases that the diagnosis diagnoses 2 potentially faulty units and one of them is truly faulty. The *Correct Type* stack shows the cases where the diagnosis does not diagnose the faulty array entry (e.g., RAT entry), but the faulty array structure (e.g., RAT) is correctly diagnosed. The *No Mismatch* stack represents cases where no misbehaved instruction is found for 30M instructions. The *Incorrect* stack shows the cases where the diagnosis process diagnoses one or more structures as faulty, none of which is the actual faulty structure. The height of each bar is normalized to all the cases on which the diagnosis procedure is invoked (i.e., all faults detected within 10M instructions as discussed in Section 4.2).

Of all detected faults, our trace-based diagnosis correctly narrows 89% of the faults down to a single non-array structure (e.g., ALU) or a specific entry in an array structure (e.g., physical register # 15) and 92% of the faults down to two structures or entries. Assuming other techniques are available for testing array structures, diagnosis only needs to narrow the fault down to the array structure (e.g., RAT) instead of the array entry (e.g., RAT entry # 10). In this case, additional 5% of the faults can be correctly diagnosed. Overall, TBFD is able to narrow 98% of the detected permanent faults down to one, or two potentially faulty structures or array entries, and the faulty array structure.

5.1. Uniquely Diagnosed Faulty Structures

When TBFD correctly narrows a detected fault down to a single unit or array entry, we categorize the fault as uniquely diagnosed. While 89% of all detected faults can be uniquely diagnosed, from Figure 4, we see that different microarchitectural structures have varying amounts of uniquely diagnosed faults.

For 5 (all except INT ALU and RAT) out of 7 structures, over 97% (up to 100%) of the detected faults are uniquely diagnosed; this shows TBFD is highly effective for diagnosing faults in these structures. In particular, virtually all the faults in Decoder can be uniquely diagnosed. This high percentage is likely due to the specific instruction word check in the first part of the diagnosis algorithm. Furthermore, over 99.6% of the ROB faults are uniquely diagnosed. This shows TBFD's meta-datapath check is essential for correct diagnoses.

For INT ALU, only 79% of the faults are uniquely diagnosed. The lower percentage is mainly due to the correlations with other structures (discussed in Section 5.2).

For RAT, however, only 45% of the faults can be uniquely diagnosed. While TBFD seems less effective for diagnosing faults in RAT, we note that without checking for faults in the meta-datapath, all of the RAT faults cannot be correctly diagnosed. Also, for array structures like RAT, there are existing testing techniques such as BIST in the processor. Thus, TBFD may not need to diagnose the fault down to a single RAT entry, as long as it identifies the RAT as the source of the fault (discussed in Section 5.3).

5.2. Non-Uniquely Identified Faulty Structures

Since the diagnosis only analyzes the faulty core's test trace and does not reconfigure the faulty core, if a correlation among two structures exists during execution, the diagnosis may not be able to uniquely diagnose the faulty component. The *Among 2* category reflects such

cases where the diagnosis diagnoses 2 suspects that are potentially faulty, with one of the suspects being the structure with the fault.

Overall, only 3% of the diagnosed faults fall into the *Among 2* category. Most of them are faults in INT ALU (18% of INT ALU faults). A closer look at the *Among 2* cases shows that all mismatching instructions that use ALU 1 always write to their registers using Reg DBus 1. It is therefore virtually impossible to separate ALU1 from Reg DBus 1 for the purpose of high-level diagnosis.

However, by narrowing down the faults down to 2 non-array structures or array entries, TBFD gives clues to where the fault may be. Then, by disabling suspected faulty components one at a time and running TBFD, the faulty unit/entry can be uniquely diagnosed. Another alternative to reduce faults in the *Among 2* category is to break the correlations among resources by explicitly changing the scheduling algorithm in the processor (proposed by Bower et. al [2]).

5.3. Faults Diagnosed in Higher Granularity

While TBFD is able to narrow down most of the faults correctly to one or two structures/array entries, only 45% of the detected RAT faults fall into *Unique* and *Among 2* categories. Such low percentage is mainly due to the reasons discussed in Section 3.4.2.

However, as BIST based techniques that test array structures are increasingly common in modern processors (for manufacturing testing), it is useful to use TBFD to diagnose the RAT (instead of a particular RAT entry) as potentially faulty and track down the actual faulty RAT entry using BIST. If we assume that it is sufficient to diagnose faults at the granularity of an array structure, TBFD can diagnose additional 44% of detected RAT faults to be in the RAT.

5.4. Undiagnosed Faults

Undiagnosed faults fall under two categories - *No Mismatch* and *Incorrect* in Figure 4. In both these cases, the diagnosis algorithm is unable to accurately attribute the location of the fault that was detected.

Of all detected faults, 2% fall in the *No Mismatch* category, where the instruction traces of the faulty and the fault-free cores do not differ. These faults may be diagnosed by collecting a longer execution trace (currently a maximum of 30 million instructions are analyzed) or by using existing deterministic replay schemes [11], [21] to re-create the fault effect that lead to its detection.

On the other hand, only 0.9% of the detected faults are mis-diagnosed by TBFD to be a fault in fault-free structures. Further, from Figure 4, we see that most of these faults are in the RAT. We observe that these RAT faults cause data corruptions and mislead the diagnosis to diagnose the datapath components as faulty. However, by disabling the suspected units and re-generating a test-trace, TBFD is more likely to diagnose these faults correctly.

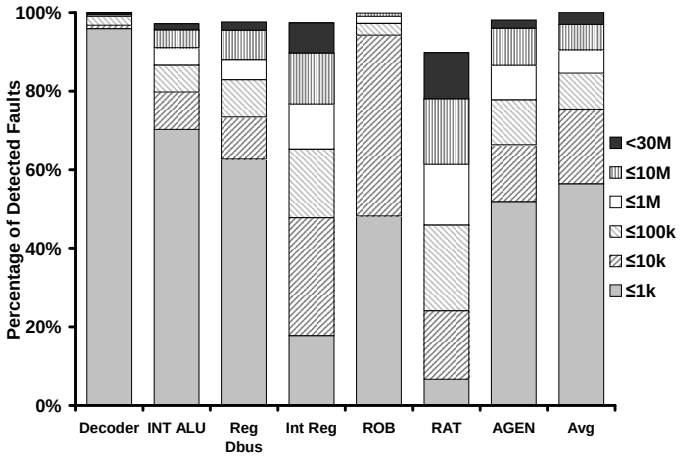


Figure 5. **Diagnosis latency in number of instructions between the start of diagnosis and the point when the fault is diagnosed. The figure shows that over 90% of the faults can be diagnosed within 1 million instructions.**

While further investigation to evaluate the best techniques to reduce, or eliminate, these undiagnosed faults is necessary to make a fool-proof diagnosis algorithm, even with these limitations, TBFD presents impressive results for microarchitecture-level fault diagnosis.

5.5. Diagnosis Latency

Besides the percentage of diagnosable faults, another metric that measures the effectiveness of our diagnosis is the latency. If the latency is too long (e.g., billions or trillions of instructions), the processors' (both the faulty and fault-free cores) down time may make TBFD unattractive when compared to other simpler techniques, such as core decommissioning.

Our simulation infrastructure does not have enough detail yet to determine the latency in terms of the execution time of the entire diagnosis module. Instead, as a proxy, we report here the latency in terms of the number of instructions that the faulty core executes between the start of our diagnosis (i.e., after the core is rolled back to the previous checkpoint) to the point where the fault is identified. Figure 5 shows this latency. The figure includes all the faults in the *Unique*, *Among 2*, and *Correct Type* categories in Figure 4.

Of all the diagnosed faults, 56% take fewer than 1k instructions and over 90% take fewer than 1M to diagnose.

From Figure 5, we see that the latency for faults in different structures varies widely. Over 99% of faults in Decoder and ROB take fewer than 1M to be diagnosed. The explicit check for front-end faults in TBFD helps shorten the diagnosis latency for Decoder faults. For ROB faults, the short latency is due to the fact that they usually cause a break in dependency and quickly lead to hardware hangs.

On the other hand, only 77% of Int Reg faults and 61% of RAT faults are diagnosed within 1M instructions. A general observation for these two types of faults is that they are activated relatively infrequently, causing fewer mismatches and taking longer for TBFD to narrow the faults down to particular unit(s).

Overall, by being able to narrow most of the faults down to one or a few locations within reasonable latency, TBFD shows that it is highly effective and incurs limited performance overhead during diagnosis.

6. Conclusions

As CMOS continues to scale according to Moore's law, hardware failures caused by permanent faults due to phenomena such as wear-out and infant morality are expected to increase. As this problem pervades the broad computing market, traditional processor-level redundancy-based solutions will be too costly to be broadly deployable. One approach is to use low-overhead fault detection techniques (which need to be "always-on"), but backed up by more expensive diagnosis techniques that need to be invoked only in the rare event of a fault.

While several microarchitecture-level schemes for detecting permanent faults have been devised in the past, fault diagnosis has been less explored. Nevertheless, fault diagnosis is crucial for tolerating permanent faults as it needs to correctly identify faulty component for repair or reconfiguration.

In this paper, we presented a diagnosis algorithm that robustly identifies faults in microarchitectural structures of different domains of a processor (front-end, datapath or meta-datapath). This technique relies on recovery-motivated checkpoint/replay mechanisms and a fault-free core in a multi-core system that can generate a fault-free trace for comparison with the faulty trace. The technique compares the faulty and fault-free execution trace, analyzes the points of differences, and reasons about the location of the fault through an intelligent diagnostic procedure.

We evaluated this diagnosis framework with fault injection experiments in a simulated system. Our results show that this approach is promising, being able to correctly identify the faulty unit in 98% of the detected faults. In 89% of the detected faults, the specific entry in an array structure was also correctly identified. Overall, with no assumptions made about detection and repair/reconfiguration mechanisms, trace-based fault diagnosis is a highly flexible framework that effectively addresses permanent fault diagnosis.

In future work, we propose to expand the diagnosis algorithm to diagnose faults in other structures both in the core and off the core. We would also like to couple this diagnosis framework with other lower-level diagnosis techniques such as BIST, to further refine the granularity at which diagnosis is performed. Finally, we are also exploring detection and diagnosis for multithreaded applications.

References

[1] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Intl. Symp. on Microarchitecture*, 1998.

[2] 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.

[3] K. Cheng and W. Lai. Instruction-Level DFT for Testing Processor and IP Cores in System-on-A-Chip. In *Intl. Design Automation Conference*, 2001.

[4] K. Constantinides et al. Software-Based On-Line Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation. In *Intl. Symp. on Microarchitecture*, 2007.

[5] M. Dimitrov and H. Zhou. Unified Architectural Support for Soft-Error Protection or Software Bug Detection. In *Intl. Conf. on Parallel Archtectures and Compilation Techniques*, 2007.

[6] M. Li et al. Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[7] M. Martin et al. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4), 2005.

[8] C. Mauer, M. Hill, and D. Wood. Full-System Timing-First Simulation. *SIGMETRICS Perf. Eval. Rev.*, 30(1), 2002.

[9] A. Meixner, M. E. Bauer, and D. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Intl. Symp. on Microarchitecture*, 2007.

[10] M. Mueller et al. RAS Strategy for IBM S/390 G5 and G6. *IBM Journal on Research and Development*, 43(5/6), Sept/Nov 1999.

[11] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Intl. Symp. on Computer Architecture*, 2005.

[12] P. Parvathala, K. Maneparambil, and W. Lindsay. FRITS: A Microprocessor Functional BIST Method. In *Intl. Test Conference*, 2002.

[13] M. Prvulovic et al. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Intl. Symp. on Computer Architecture*, 2002.

[14] P. Racunas et al. Perturbation-based Fault Screening. In *Intl. Symp. on High Performance Computer Architecture*, 2007.

[15] S. Sahoo et al. Using Likely Program Invariants to Detect Hardware Errors. In *Intl. Conf. on Dependable Systems and Networks*, 2008.

[16] S. Shyam et al. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[17] D. Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Intl. Symp. on Computer Architecture*, 2002.

[18] Virtutech. Simics Full System Simulator. Website, 2006. http://www.simics.net.

[19] N. Wang and S. Patel. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3), July-Sept 2006.

[20] E. Weglarz et al. Testing of Hard Faults in Simultaneous Multithreaded Processors. In *Intl. Online Test Symp.*, 2004.

[21] M. Xu, R. Bodik, and M. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Intl. Symp. on Computer Architecture*, 2003.