SWAT: An Error Resilient System

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

To this end, we are developing SWAT (SoftWare Anomaly Treatment) – a low-cost reliability solution that effectively handles multiple sources of faults by detecting anomalous software behavior. At the last SELSE, we motivated SWAT and presented a preliminary detection component that detects hardware failures by monitoring simple software level symptoms. This paper presents two significant enhancements to the SWAT system over the last year: (1) an effective diagnosis strategy that identifies the faulty microarchitectural unit by exploiting a checkpoint/replay based recovery mechanism and analyzing the faulty core's instruction trace, and (2) a sophisticated detection mechanism that specifically targets silent data corruptions by using compiler-inserted range-based invariants to further improve detection coverage and latency. The detection strategy leverages the online diagnosis strategy in a novel way to enable aggressive use of invariants while minimizing the impact of false positives at runtime.

I. Introduction

Hardware reliability is becoming a major obstacle to reaping the benefits of increased integration projected by Moore's law. With increased scaling, this problem continues to worsen and threaten the failure of shipped components due to several reasons including aging or wear-out, infant mortality, soft errors, design defects, process variations, and so on [2]. Such a scenario requires mechanisms to detect, diagnose, recover from in-field failures, and possibly repair/reconfigure around these failed components so that the system can provide reliable and continuous operation.

The reliability challenge today pervades the entire computing market – from high-end servers that tolerate high overheads for reliability to low-end desktop machines that are stringent about overheads for economical reasons. Thus, a reliability solution that can be effectively deployed in the broad market must incur limited overheads in area, performance, and power. A SELSE-2 industry panel converged on a 10% area overhead target to handle all sources of chip errors as a guideline for academics. In this context, traditional solutions that involve heavy redundancy, such as dual modular redundancy or triple modular redundancy, are no longer viable. Solutions such as redundant multithreading and its various flavors improve on this, but still incur large overheads in performance and/or power [9].

We make two high-level observations that motivate our quest for low-overhead reliability solutions. First, the reliability solution needs to handle only device faults that propagate through higher level of the system and become observable to the software. Second, despite this impending reliability threat, fault-free operation remains the common case and must be optimized, possibly at the cost of increased overhead after a fault is detected (in accordance with Amdahl's law).

These observations motivate our **SWAT** (**S**oft**W**are **A**nomaly **T**reatment) system where faults are detected by watching for anomalous software behavior, or symptoms of faults, using zero to low-cost hardware and software monitors [6]. Such a strategy treats hardware faults analogous to software bugs, potentially leveraging solutions for software reliability to further amortize the overhead. After an error is detected, the diagnosis process is invoked to identify the error type (transient or permanent error) and the faulty component (if a permanent error is diagnosed). Based on the result of the diagnosis, the relevant recovery and/or repair is performed to circumvent the error.

In previous work, to investigate the feasibility of the SWAT approach, we explored using low-cost always-on monitors of software anomalies, called *symptoms*, for detecting hardware faults [5][6]. The symptoms we used were *fatal traps* from either the application or the OS (indicated by the hardware), *application aborts* (indicated by the OS), *hangs* of either the application or the OS (indicated by a hardware hang detector), and *high OS activity* (indicated by the hardware per-

This work is supported in part by an IBM faculty partnership award, 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, Sun Microsystems, Inc's OpenSPARC Centers of Excellence at the University of Illinois at Urbana-Champaign, and an equipment donation from AMD.

formance counter). Through microarchitecture-level fault injections into 8 hardware structures in a simulated superscalar out-of-order processor, we found that 95% of unmasked faults in 7 of the 8 structures resulted in detectable symptoms within 10 million instructions of simulation. Additionally, only 0.8% of these faults resulted in silent data corruptions (SDC). These results indicate the effectiveness of these symptoms, in spite of their simplicity, and motivate the SWAT strategy.

In this paper, we present two significant enhancements to the SWAT framework:

- We derive a diagnosis strategy that identifies the faulty microarchitectural component by exploiting checkpoint/replay based recovery mechanisms and by analyzing the faulty core's instruction trace. Of all the detected faults, our approach correctly identifies the faulty component in 96% of the cases.
- We explore a sophisticated detection mechanism that uses compiler-inserted range-based invariants to detect hardware faults. When used along with the simple symptoms described above, this scheme further improves the coverage, shortens the detection latency, and reduces the number of total SDC events by about 50% with low overheads.

With an ever increasing list of reasons for hardware failures, systems that detect hardware failures through anomalous software executions, similar to SWAT, become more attractive owing to their low overhead and their ability to deal with multiple failure sources. The designs for diagnosis and detection presented here form the cornerstones for building such error resilient systems.

II. Fault Diagnosis

Since SWAT performs detection by observing anomalous software behavior, it can detect both hardware faults and software bugs. Consequently, the SWAT diagnosis component is thus of paramount importance to not only distinguish between faults in hardware from faults in software, but in the case of hardware faults to also identify the type (transients or permanents) of hardware faults to facilitate the appropriate recovery and/or repair.

In the SWAT diagnosis framework, we assume a multicore system where a fault-free core is available. We also currently assume that all detections are because of faults in the core. Upon detection, we use a repeated rollback/replay strategy from a checkpoint to distinguish transient faults, permanent faults, and software faults [6]. The diagnosis firmware observes whether the symptom recurs after the execution is replayed from the checkpoint on the symptom-causing core. A lack of symptom indicates that a transient fault might have occurred and is appropriately recovered. If the symptom recurs, the execution is replayed on another fault-free core. If the symptom recurs in the fault-free core, a software fault is diagnosed and propagated to the software layer. A lack of symptom in the fault-free core indicates the existence of a permanent fault in the original core. While transient hardware faults in the original core can be dealt with by a simple re-execution, permanent hardware faults need more sophistication to enable reconfiguration/repair to prevent further activation of the fault.

A. Overview of Trace-Based Fault Diagnosis

When a permanent fault is diagnosed in a core, the simplest solution is to decommission the entire core to prevent further corruption of the system. However, such an approach may be too conservative as most of the core may be fully functional and the core may be repairable by deconfiguring only the faulty microarchitectural modules (e.g., a physical register). Motivated by this observation, we employ a microarchitecture-level fault diagnosis scheme called Trace-Based Fault Diagnosis (TBFD).

As the name suggests, TBFD identifies permanent faults by analyzing instruction traces. More specifically, it compares the execution trace of the faulty core to that of the fault-free core. When a mismatch occurs between the result generated by the two cores, TBFD reasons that the corresponding instruction in the faulty core has activated the fault and begins to track down the faulty microarchitectural module.

TBFD proceeds as follows. When a faulty core is identified, the faulty core rolls back to a previous checkpoint and generates an instruction trace that is tagged with microarchitectural resources used by each instruction (called faulty execution trace). To generate golden states for comparisons, a fault-free core is loaded with the same checkpoint and produces the fault-free execution trace. A mismatch between the result of the faulty execution and that of the fault-free execution invokes the TBFD algorithm to systematically track down the faulty module.

Currently, TBFD assumes a superscalar processor with register-renaming and targets faults in three main areas of the processor core: the front-end (e.g., the decoder), the meta-datapath (the datapath that tracks dependences among instructions, including the RAT and ROB), and the datapath (e.g., functional units, physical registers, etc.). When a mismatch occurs, TBFD first checks the decoded information (opcode, source operands, and destination operands) of the faulty instruction. If this mismatches with that of the fault-free execution, a fault is suspected in the front-end. Else, using the faulty execution trace, TBFD checks the logical-to-physical register mappings used by the faulty instruction to verify the integrity of the meta-datapath. One common symptom of a meta-datapath fault is that a physical register is mapped to two or more logical registers. If the register mappings are correct, a fault is suspected in the datapath.

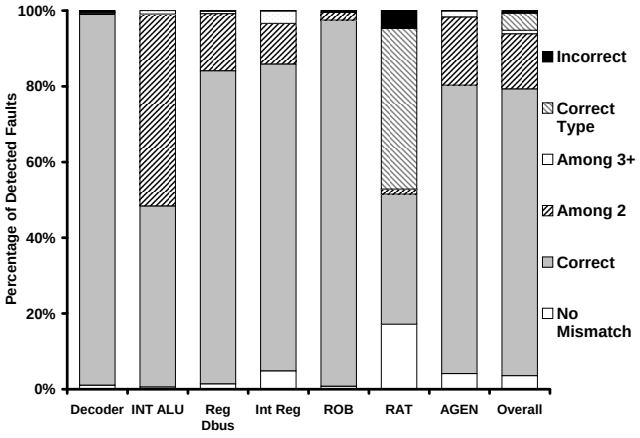

Fig. 1. **Effectiveness of microarchitecture-level fault diagnosis. The figure shows the ability of the diagnosis algorithm to accurately diagnose detected faults. On an average, 96% of the detected faults are accurately diagnosed.**

Since a faulty instruction could have used multiple resources in each of the areas described above (e.g., an add instruction may use ALU0, data-bus1, and physical register 20), TBFD uses counters to track potentially faulty structures (similar to the approach in [3] but the counters are in software and are invoked only after a fault is detected). When a larger number of faulty instructions are encountered, the counter value of the faulty module will be higher than other modules and thus can be identified.

B. Effectiveness of Trace-Based Fault Diagnosis

To evaluate TBFD, 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 simulator [11]. Together, these simulators provide cycle-by-cycle microarchitecture-level timing simulation of a real workload (6 SpecInt2000 and 4 SpecFP2000) running on a commercial operating system (full Solaris-9 on SPARC V9 ISA) on a modern out-of-order superscalar processor and memory hierarchy.

We inject 11,200 permanent faults into 7 microarchitectural structures of the simulated processor. For the injected faults that are detected using our symptoms [6], TBFD is invoked to identify the faulty microarchitectural structure.

Figure 1 presents the results indicating the effectiveness of the diagnosis for faults in different microarchitectural structures. In each bar, the *No Mismatch* stack represents cases that the faulty core's trace is identical to the golden core's trace within 10M instructions. The *Correct* stack represents cases that the diagnosis process correctly and uniquely identifies the faulty unit or the faulty entry within an array structure in the faulty core. The *Among N* stack represents cases that the diagnosis process identifies N potential faulty units and the faulty structure is one of the identified units. The *Correct Type* stack shows the cases where the diagnosis does not identify the specific faulty array entry (e.g., RAT entry) but the faulty array structure (e.g., RAT) is correctly identified. The *Incorrect* stack shows the cases where the diagnosis identifies one or more structures as faulty, none of which is the actual faulty structure.

Of all detected faults, our trace-based diagnosis is able to correctly and uniquely identify 76% down to the single faulty unit or array entry. Further, for 90% of the detected faults, the faulty unit falls within the two units reported by TBFD as potentially faulty. These results show that TBFD is effective in identifying most of the faults down to an array entry at the microarchitecture-level.

For RAT, we found that TBFD cannot identify many of the faults down to an array entry because speculative instructions could cause a live physical register to be freed and then get squashed. Since TBFD only analyzes retiring instructions, the original faulty RAT entry is harder to track. However, if more traditional diagnosis techniques are available, it is sufficient for TBFD to identify faults at the granularity of the array structure. Given this assumption, an additional 42% of the detected RAT faults can be identified (*Correct Type* stack) down to the RAT structure.

Overall, by collecting and analyzing the instruction trace from the faulty execution, TBFD correctly narrows 96% of the detected faults down to a single array entry (*Correct*), 2 and 3 faulty units/array entries (*Among 2 and Among 3*), and the array structure (*Correct Type*).

These results show that TBFD is effective in diagnosing permanent faults at the microarchitecture-level, without (1) assuming specialized architecture (such as DIVA [1]), (2) changing the instruction scheduler within the processor, and (3) disabling suspected faulty units and retrying as in prior work [3]. This approach also further enhances the diagnosis capability of the SWAT system.

III. The SWAT-I Invariant-Based Detection Framework

In our previous implementation of the detection module in the SWAT system, we used simple software-observable events to infer the presence of an underlying hardware fault [6]. Although these detection mechanisms incur negligible hardware overheads for detection and result in highly competent coverage, more sophisticated detection mechanisms must be used to further improve this coverage and latency by reducing the faults that escape detection in the simulated 10M instruction window and eventually reduce SDC events. For this purpose, we designed the SWAT-I detection framework, an enhancement to the proposed SWAT detection mechanisms [6] that explores the use of program invariants.

Program Invariants are program properties involving program values/attributes at some particular program point that are expected to hold on all possible inputs, i.e., they are sound invariants. Likely Program Invariants are program properties involving program values that hold on many executions on different inputs and are expected or *likely* to hold on other inputs. Extraction of likely program invariants is easier than extraction of sound invariants, as we do not need costly static analysis methods to prove program properties. The extraction may be done either online or offline. With compiler support, invariants can be extracted offline, and transparently, during development/testing phases.

Invariants are broadly classified into three broad categories: Value-based, Control-flow-based and PC-based invariants. In the current SWAT-I system, we only use range-based invariants (a particular type of value-based invariant), which specify a range with constant lower and upper bounds for specific program values. For example, a sample range invariant on a program variable x will be of the form [MIN, MAX], where MIN and MAX are constants inferred from offline training such that $MIN \leq x \leq MAX$ is true for all the training runs. As a first step, we decided to use range-based invariants, as they can be easily and efficiently generated. They are also much easier to enforce within checking code and cause many fewer false positives compared to most other types of invariants.

Since we propose to use likely program invariants to detect permanent faults, one limitation is that some of these invariants may be *false positives*. An invariant is called a false positive for some particular input, if it does not hold true for that input. To handle the false positives with minimal overhead, we need an efficient method for detecting them online, unlike transient hardware faults where relatively low-cost techniques such as pipeline flush can deal with false positives [8]. In SWAT-I, we leverage the rollback/replay support in the diagnosis framework and detect false positives by rolling back and replaying the execution on the fault-free core when an invariant violation is detected. We limit the overhead caused by these rollbacks by limiting each invariant to cause at most one rollback and replay. If the rollback and replay determines a false positive, we disable this invariant for future executions.

A. Generating Invariants and Invariant Checking Code

Our SWAT-I framework has two distinct components: invariant generation and invariant insertion. Both of these use the LLVM compiler infrastructure [4].

For the invariant generation phase, we use compile-time instrumentation to monitor program values online. In this work, we monitor only the store values of all integer types (both signed and unsigned) of size 2, 4, and 8 bytes as well as floating point types, to keep the overhead low. For the invariant insertion phase, we insert calls to appropriate invariant checking code through another compile-time instrumentation pass and generate native code for SPARC-based Solaris system using the *Sun cc* compiler.

B. Effectiveness of Invariants for Detection

The simulation infrastructure used to evaluate invariants is identical to that used for evaluating the diagnosis framework. We evaluated our invariant-based approach in conjunction with the four low-cost detection mechanisms (fatal traps in application, fatal traps in OS, application aborts and high-OS activity) built into the base SWAT system [6] using five different metrics. For the experiments, we used five SPEC benchmarks: four SpecINT benchmarks (gzip, bzip2, mcf, parser) and one SpecFP benchmark (art). Currently, we use 12 training inputs collected from external sources or generated through a script (including *test* and *train* inputs) for the invariant generation phase for all applications. We inject 5600 permanent faults into 7 microarchitectural structures of the simulated processor.

For the evaluation of the effectiveness of invariant-based approach, *ref* input was used to compute five different metrics: *Coverage*, *Latency*, *False positives*, *SDCs* and *Overhead*. We used two configurations of the timing simulator for the experiments: *12-input* that enforces invariants and *baseline* that does not. We instrument the application with invariants checking code for both configurations, but disable the enforcement of invariants in the simulator for the baseline case. This step is necessary to minimize the differences in the binaries and obtain a valid coverage comparison between the two cases because the behavior of faults depends on both the static code layout and the dynamic instruction sequence. We also kept the invariant checks for the completion runs (needed to compute SDCs) of the 12-input case but removed the false positive invariants as we did not have false positive detection support in the functional simulator used for the completion runs. When invariant violations occur during the completion runs, the applications exit.

1) Coverage: We define *Coverage* as the total number of detections as a percentage of total non-masked faults (i.e. total number of fault injections excluding those cases which are masked architecturally or at the application level).

For each fault injection, we run the full timing simulation for 10 million instructions. Figure 2 partitions all the fault injections into eight categories based upon the detection method which detects the fault within the 10M instruction window for both configurations: *12-input* and *baseline*. The seven categories are as follows. *Arch-Mask* represents the set of injections that were architecturally masked. *Appl-Mask* represents the set of faults which

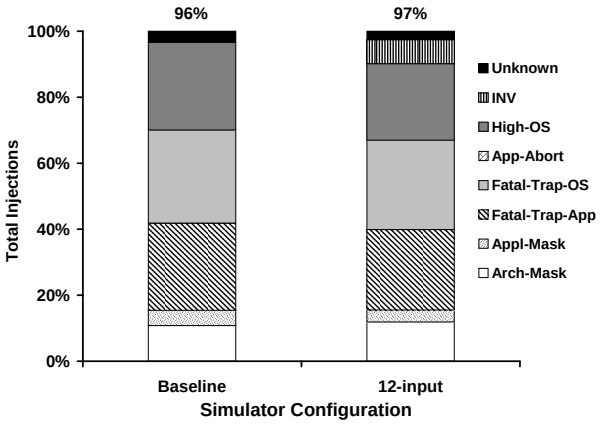

Fig. 2. **Effectiveness of different symptoms to detect faults in various microarchitecture structures. Software-level invariants improve the coverage of SWAT detection from 96% to 97%.**

are masked at the application level. The topmost stack shows the set of non-masked fault injections that remain undetected after the simulated 10M instruction window (we call this category *Unknown*). The other partitions — *INV*, *High-OS*, *App-Abort*, *Fatal-Trap-OS*, and *Fatal-Trap-App* — represent detections by the corresponding techniques [6], where *INV* represents detections through the invariant method. The faults not detected are categorized under the *Unknown* category. The number above each bar shows the total coverage for each version.

Three points are apparent from this graph. First, the invariant detection detects nearly 10% of faults. Second, the invariant detection detects some faults that the baseline techniques missed, increasing the total coverage from 96% to 97%. Third, the invariant detection detects some faults that would have been caught by other techniques (about 8% of total faults). Overall, we see a *23%* reduction of unknown cases from 189 in the baseline version to 145 in the 12-input version.

2) Latency: The detection *Latency* is the total number of instructions retired from the first architecture state corruption (of either OS or application) until the fault is detected by one of the above techniques. For the latency experiments, we used two different configurations of the simulator, as in the coverage experiments.

The total number of detections with latency < 100K, as a percentage of the total number of non-masked faults, increases from 79% to 81%. Since hardware checkpointing can handle recovery at this detection latency [10], using invariants increases the number of faults amenable to hardware recovery, making recovery simpler. Faults detected at a latency larger than 100K instructions will require software checkpointing schemes.

3) False Positives: We define *False positive rate* as the fraction of all the static invariants that do not hold true for some particular input. We used all 12 training inputs for generating invariants. To evaluate the false positive rate, we used the *ref* input. With 12 inputs, we found the false positive rate to be less than 5% for all the applications and 0% for three of the applications. As the false positive rate was sufficiently low for our purpose, we did not need to use more inputs. Quite surprisingly, after just two inputs, nearly 50% of the invariants are true positives. This low rate of false positives from many inputs motivates the use of likely invariants for detecting permanent faults.

4) Silent Data Corruptions (SDCs): In order to determine the SDC events among the *unknown* cases after 10M instructions of detailed timing simulation with fault injection, we ran the application to completion using a functional simulator. If the application failed to terminate within five hours of simulation, then we declared the result an application or OS hang. We simulate both *12-input* and *baseline* versions to completion.

Overall, the invariants are able to reduce the SDC events by almost **50%** from **45** events to **23** events. We consider the reduction in the SDC events as the most important contribution of the invariants. Though some SDC events remain, we believe that the use of more sophisticated invariants can make the number of SDC events negligible.

5) Overhead: For evaluating the overhead of the invariant checking code, we used two versions of the application code - one with invariants checking code and the original code without any invariants check. We kept all the invariant checks including false positives for evaluation of overhead. We first evaluated the overhead with a 1GHz Sun UltraSPARC-T1 processor. The average overhead measured as the geometric mean of overhead of all benchmarks is around 14%. The higher than expected overhead is most likely due to the simple (1-wide, 6-stage pipeline) architecture, which is unable to hide the latency of extra instructions and cache misses. To verify this, we ran the same overhead experiments on the x86 architecture, a dual processor 1755 MHz AMD Athlon^{TM}MP 2100+ machine. The average overhead (geometric mean) is only 5%, which makes this initial approach promising.

IV. Conclusions

With ever increasing hardware overheads to meet required reliability targets, systems like SWAT that detect hardware failures by tracking anomalous software behavior become increasingly attractive. In addition to effective low-cost detection, accurate diagnosis to narrow down the faulty hardware component is of prime importance to initiate repair/reconfiguration of the faulty hardware.

This paper presented two important components of the SWAT system: (1) a microarchitecture-level diagnosis framework, called TBFD, that performs fault diagnosis by analyzing the faulty execution trace with the help of a fault-free core and (2) a detection framework,

called SWAT-I, that uses compiler-inserted invariants to improve detection coverage and latency with acceptable overheads. These components are essential for the SWAT system to ensure continuous and reliable operations. In addition to improving the ability of these components to detect and diagnose faults, we are also working on other components of SWAT and working towards the goal of providing a low-cost full-system reliability solution for future hardware.

REFERENCES

[1] T. M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 1998.

[2] S. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *IEEE Micro*, vol. 25, no. 6, 2005.

[3] F. Bower *et al.*, "A Mechanism for Online Diagnosis of Hard Faults in Microprocessors," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2005.

[4] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, 2004.

[5] M. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou, "Towards a Software-Hardware Co-Designed Resilient System," in *3rd Workshop on Silicon Errors in Logic - System Effects*, 2007.

[6] M. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[7] M. Martin *et al.*, "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *SIGARCH Computer Architecture News*, vol. 33, no. 4, 2005.

[8] P. Racunas *et al.*, "Perturbation-based Fault Screening," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[9] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," in *Proceedings of International Symposium on Fault-Tolerant Computing (FTCS)*, 1999.

[10] D. Sorin *et al.*, "Fast Checkpoint/Recovery to Support Kilo-Instruction Speculation and Hardware Fault Tolerance," Computer Sciences Department, University of Wisconsin, Madison, Tech. Rep. 1420, 2000.

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