Appears in Proceedings of the 38^{th} International Conference on Dependable Systems and Networks (DSN), June 2008.

Using Likely Program Invariants to Detect Hardware Errors [*]

Swarup Kumar Sahoo, Man-Lap Li, Pradeep Ramachandran,
Sarita V.Adve, Vikram S. Adve, Yuanyuan Zhou

Department of Computer Science
University of Illinois at Urbana-Champaign
swat@cs.uiuc.edu

Abstract

In the near future, hardware is expected to become increasingly vulnerable to faults due to continuously decreasing feature size. Software-level symptoms have previously been used to detect permanent hardware faults. However, they can not detect a small fraction of faults, which may lead to Silent Data Corruptions(SDCs). In this paper, we present a system that uses invariants to improve the coverage and latency of existing detection techniques for permanent faults. The basic idea is to use training inputs to create likely invariants based on value ranges of selected program variables and then use them to identify faults at runtime. Likely invariants, however, can have false positives which makes them challenging to use for permanent faults. We use our on-line diagnosis framework for detecting false positives at runtime and limit the number of false positives to keep the associated overhead minimal. Experimental results using microarchitecture level fault injections in full-system simulation show **28.6%** *reduction in the number of undetected faults and* **74.2%** *reduction in the number of SDCs over existing techniques, with reasonable overhead for checking code.*

1. Introduction

As CMOS feature sizes continue to decrease, hardware reliability is emerging as a major bottleneck to reap the benefits of increasing transistor density in microprocessor design. Chips in the field are expected to see increasing failure rates due to permanent, intermittent, and transient faults, including wear-out, design defects, soft errors, and others [2]. The traditional approach in microprocessor design of presenting an illusion of a failure-free hardware device to software will become prohibitively expensive for commodity systems. Traditional solutions such as dual modular redundancy for tolerating hardware errors incur very high overheads in performance, area and power. Recent hardware solutions such as variations on redundant multithreading improve on this, but still incur significant overheads [27].

Recently, researchers have investigated using *software-visible symptoms* to detect hardware errors [5, 9, 19, 24, 25, 26, 30, 32]. While much of that work focuses on transient or intermittent faults that last a few cycles (e.g., 4 cycles or less), we have explored using these symptoms to detect permanent faults in hardware [9].

Using software-level symptoms to detect permanent faults in hardware has several benefits over traditional hardware-level solutions. First, using software-level symptoms deals with only those errors that actually affect software correctness. The rest of the faults are safely ignored, potentially reducing the incurred overhead due to detection and recovery. Second, the reliability targets for the system under consideration dictates the overheads that the system allows to achieve those targets. Using software-level symptoms for detection facilitates exploring these trade-offs in reliability and overhead seamlessly as they are highly customizable.

We proposed a system design called SWAT, a firmware-level low-overhead reliability solution that could potentially handle multiple sources of hardware failures [9] using software symptoms such as fatal hardware traps, software hangs, abnormal application execution, and high OS activity. Implementing these detectors in a thin firmware layer would present significantly lower hardware cost than using traditional circuit-level hardware detectors. These detectors help identify over 95% of hardware faults in many structures. Additionally, 86% of these detections can be recovered using hardware checkpointing schemes, while all these detected faults are software recoverable [9].

Nevertheless, using these simple symptoms as detectors results in an SDC rate of 0.8% for permanent hardware faults in the current SWAT system, which may not be acceptable for most systems. This motivates the use of more sophisticated detectors to further reduce this SDC rate and increase detection coverage. In addition, using more sophisticated detectors has the potential of reducing the detection latency of the detected faults, making more faults amenable to hardware recovery. Recovery through hardware checkpointing techniques, which can treat detection latencies upto 100K

[*] 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.

cycles [28], are more attractive than those that use software checkpointing techniques for recovery as it facilitates seamless recovery of both the application and the OS in the event of a fault with much lesser overhead.

In this work, we extend the set of symptom-level detectors in SWAT to include program-level invariants that are derived from program properties observed during program execution. We use "likely program invariants" which have been shown to be a powerful approach in detecting software bugs [4, 6]. We derive likely program invariants by monitoring the execution of a program for different inputs and identifying program properties that hold on all such executions. [1] A major drawback with using likely invariants for error detection is that they may lead to *false positives*: some of the inferred program invariants may be violated for an input as the program behavior on that input is different compared with the training inputs used to extract invariants. Hence, likely program invariants have been proposed and used primarily for analysis purposes such as program evolution [4], program understanding [7], and detecting and diagnosing software bugs [4, 6, 12, 33, 13]. The only exceptions have been for detecting transient hardware faults, where a false positive can be identified quickly and cheaply [22, 24, 3].

In this paper, we propose and evaluate a hardware-assisted methodology to use likely invariants for detecting permanent (or intermittent) hardware errors safely. The SWAT system has a hardware-assisted diagnosis framework and we adapt it to detect false positives at runtime. We also limit the number of false positives in a novel way to keep the associated overhead due to false positive detection low. Using the principles discussed above, we designed the iSWAT framework for invariant detection and enforcement, and we implemented it as an extension of the SWAT system [9].

The contributions of this work are:

- We demonstrate a new hardware-supported strategy for using unsound program invariants for detecting permanent hardware errors. We believe this is the first work to use unsound invariants for such errors.
- We show that likely invariants can be extracted efficiently in software for realistic programs, unlike previous work which used only toy benchmark programs [22]. Furthermore, because of our tolerance for false positives, we only need 12 inputs for extracting our invariants while others have used hundreds of inputs [4, 22].
- We provide a realistic and comprehensive evaluation with full-system simulation by injecting faults into different micro-architectural structures. Such faults present more realistic fault scenarios than the previously studied application-level fault injections.
- The most important outcome from our experiments is that our technique reduces SDCs by 74.2%: fewer than 0.2% of all fault injections are now SDCs.

In more detail, our experimental results show that the number of undetected faults in iSWAT decreases by nearly **28.6%** compared with the base SWAT System. The number of SDCs reduces from **31** to **8** (i.e., 74.2% reduction). The number of detections that are hardware recoverable (with latency less than 100K instructions) improves slightly by 2%. The mean overhead due to invariant checking code is low - 14% on an UltraSparcIIIi machine and only 5% on an AMD Athlon machine. Moreover, this work is just a first step using one simple style of invariants. These results show that using likely invariants is a promising way to improve overall reliability, at a low cost.

The rest of this paper is organized as follows. Section 2 provides a brief overview of likely invariants. In Section 3, we describe the iSWAT System in detail, explaining how we exploit the diagnosis module to detect false positives caused by the invariants. Section 4 discusses the evaluation methodology, the results of which are discussed and analyzed in Section 5. Related work is discussed in Section 6. Section 7 draws conclusions and implications from our experience with the iSWAT framework and discusses future work.

2. Invariant-Based Error Detection

In this section, we provide some background on likely program invariants and then discuss the particular type of likely invariants we use to detect permanent faults.

2.1 Likely Program Invariants

A program invariant at a particular program point P is a property that is guaranteed to hold at P on all executions of the program. Static analysis is the most common method to extract such sound invariants. A combination of offline invariant extraction pass and static analysis, or theorem proving techniques, has also been suggested to extract sound invariants [20]. However, current techniques are not scalable enough to generate sound invariants for real programs [20]. Also, they can not identify algorithm-specific properties that are not explicit in the code (e.g. some inputs are always positive).

Likely Program Invariants are properties involving program values that hold on many executions on all observed inputs and are expected to hold on other inputs. However, they are unsound invariants which may not hold on some inputs. Extracting likely program invariants is easier than extracting sound invariants as we do not need expensive static analysis methods to prove program properties and can identify algorithm specific properties. The extraction can be done either online or offline. In online version, invariants are extracted and used during program execution in the production runs. Online extraction can present unacceptable overheads

[1] With simple compiler support, this "training" phase can be performed transparently during debugging runs for any program, and could even be extended into production runs with more sophisticated tools [11].

to program execution, and may in fact be infeasible without hardware support. The offline version, on the other hand, extracts invariants in a separate pass during program testing or debugging, and these generated invariants can be used later during the production runs. During the testing phases of software development, the extra overhead of invariants extraction can be tolerated. This makes offline invariant extraction a powerful method, allowing the use of more complex invariant mining techniques than would be feasible with online methods. With compiler support, this "training" phase can be done transparently at development time.

We can broadly classify likely program invariants into three categories. *Value-based* invariants specify properties involving only program values, and can be used for a variety of tasks including software bug detection, program understanding and program refactoring etc [4, 11, 7, 12, 6]. *Control-flow-based* invariants specify properties of the control flow of the program, and have been used previously to detect control-flow errors due to transient faults [30, 29, 5]. *PC-based* invariants specify program properties involving program counter values, and have been proposed for detecting memory errors in programs during debugging [33].

2.2 Range-Based Invariants

One of our main goals in exploring the use of invariants to detect permanent faults is to improve the coverage of detection and reduce the number of SDCs. Since SDCs are typically caused by erroneous values written to output, we explore the use of value-based invariants to detect permanent faults. The other two class of invariants can detect control-flow or memory errors, which generally result in anomalous software behavior that can be detected by the other detectors in SWAT. For example, erroneous control-flow typically results in a crash which can be caught by the *FatalTrap* symptom in SWAT [9]. In contrast, we expect value-based invariants to capture deviations of values that do not result in any significant change of program behavior to cause an application or OS crash, but may still result in incorrect output.

As a first step towards using likely program invariants for permanent hardware faults, we use a particular form of value-based invariants known as range-based invariants. A range-based 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.

These range-based invariants are suitable for error detection for various reasons. These types of invariants can be easily and efficiently generated by monitoring program values. They are also composable – the invariants can be generated for each training input separately and can then be combined together to generate invariants for the complete training set. These invariants are also much easier to enforce within the checking code compared to other forms of invariants as they are simple and involve a single data value. In the ongoing future work, we are exploring a broader class of invariants.

3. The iSWAT Detection Framework

We implement the above described range-based invariants as an extension to the existing SWAT System [9] to build the iSWAT system that uses likely program invariants as an additional software-level symptom to detect hardware faults.

3.1 Overview of SWAT System

The SWAT system uses low-overhead software-level symptoms to detect the presence of an underlying hardware faults, and exploits a firmware-assisted diagnosis and recovery module to recover the system from multiple sources of faults [9]. While this paper targets permanent hardware faults, our methods, similar to SWAT, also extend readily to detect transient faults.

SWAT assumes a multi-core architecture under a single fault model where a fault-free core is always available. The system also assumes support for a checkpoint/rollback/replay mechanism and a *firmware layer* that lies between the processor and the OS to monitor and control such mechanisms.

Detection: SWAT uses four low-cost symptom-based detection mechanisms that require little new hardware or software support. These mechanisms look for anomalous software execution as symptoms of possible hardware faults. We briefly describe them below; the details can be found in [9].

1. *FatalTrap*: Fatal hardware traps are those traps (caused by either the application or the OS) that do not occur during fault-free execution. In Solaris, some of the fatal traps are RED (Recover Error and Debug) State trap, Data Access Exception trap etc.

2. *Abort-App*: These indicate instances of a segmentation fault or illegal operation, when the OS terminates the application with a signal. In such cases, the OS informs the detection framework that the application performed an illegal operation, leading to a detectable symptom.

3. *Hangs*: Application and OS hangs are other common symptoms of hardware faults [19]. SWAT uses a low hardware-overhead heuristic hang detector, based on (offline) application profiling to detect hangs with high fidelity.

4. *High OS activity*: In normal executions, on a typical invocation of the OS, control returns to the application after a few tens of OS instructions, except cases such as timer interrupt or I/O system calls. As a symptom of abnormal behavior, SWAT looks for instances of abnormally high contiguous OS instructions to indicate the presence of an underlying fault.

Diagnosis: After the fault detection, the diagnosis framework is invoked to distinguish the source of the fault as a software fault, or a transient fault, or a permanent fault. The diagnosis component rolls back to the last checkpoint and replays the execution on the original core. If the symptom does not recur, it infers a transient fault and continues execution. However, if the symptom recurs, it re-executes on another core (assumed to be fault-free) to distinguish software faults (in which case the symptom would most likely reoccur on the fault-free core) from permanent hardware faults (in which case the symptom would not occur). We may need to do the replay multiple times on the two cores to distinguish not-deterministic software errors. In the case of a permanent fault, the diagnosis module also does microarchitecture level diagnosis to identify the faulty microarchitectural structure [10].

Recovery and Repair: For recovery, SWAT assumes some form of checkpoint/restart mechanism that periodically checkpoints the state of the system. Depending upon the requirements, appropriate hardware checkpointing or software checkpointing mechanisms, or a combination of both, can be used to recover the system after detection. SafetyNet [28] and Revive [18, 23] show reasonably low overhead for hardware checkpoint/replay. In the event of a permanent hardware fault, the component that is diagnosed as faulty can be reconfigured or disabled.

3.2 The iSWAT system

The iSWAT system extends the above described SWAT system to include the violation of likely program invariants as possible symptoms that indicate the presence of a hardware fault. These invariants are derived from “training” runs of the application and the invariant checking code is embedded into the application. iSWAT exploits the diagnosis and recovery module of the SWAT system to detect and disable false-positive invariants at run-time.

3.2.1 Generating Invariants and Invariant Checks

The iSWAT system leverages support from the compiler for two distinct components: invariant generation and invariant insertion. Both of these use the LLVM compiler infrastructure [8].

Invariant Generation: We use compile-time instrumentation to monitor program values during training runs in order to generate likely program invariants. We can monitor many different types of values, including load, store, return and intermediate result values. For this work, we decided to monitor only the store values as checking values stored to memory has the most potential to catch faults, as all necessary computations eventually pass their results to stores. Also, monitoring only the stores helps us keep the overhead of detection low. We monitor stored values of all integer types (both signed and unsigned) of size 2, 4, and 8 bytes as well as single and double precision floating point types. We do not monitor integer stores of size 1 byte (character data types), as they represent only a small range of values and hence may be ineffective to detect faults.

We do the code generation for invariant generation in two steps. In the first step, we use LLVM-1.9 with llvm-gcc-3.4 to generate the LLVM bytecode and run an instrumentation pass to insert calls to monitor the store values. Then we generate a C program from the LLVM bytecode file through the LLVM C back end. Finally, we generate SPARC native code through the *Sun cc* compiler. We use the generated program to create the invariants for each input separately, which we then combine to form the final invariants in another offline pass.

Invariant Insertion: The invariants generated by the offline pass then need to be inserted into the code to check the values being stored. For this, we take the invariant ranges from the generation phase and then insert calls to the invariant checking code at the LLVM byte-code level through another compile-time instrumentation pass. Then, as in the invariant generation phase, we generate a C program from LLVM bytecode through LLVM C back end and finally, use *Sun cc* compiler to generate native code.

3.2.2 Handling False-Positive Invariants

False positives present a major short-coming for likely program invariants as detecting them may incur high overheads in the presence of permanent faults. For transient hardware faults, relatively low-cost techniques such as pipeline flush can help to tolerate false positives [24]. If the fault occurs again after pipeline flush, the invariants will be updated and this is done cheaply using hardware support. In contrast, for a framework like iSWAT System that supports permanent fault detection, more expensive rollback and replay on the same and a fault-free core is needed to detect false positives. Too many false positive detections may thus lead to exorbitant overheads.

While training with too many inputs can potentially make false positives rare, the ranges may become too broad rendering the invariants ineffective for error detection. In our framework, we propose to train with a set of inputs such that the false positive rate is sufficiently low. In general, the number of false positives can be used to guide how many inputs to use for training.

iSWAT leverage the existing diagnosis framework in the SWAT system [10] to detect the remaining false positives at run-time. In the event of an invariant violation, we follow the full rollback and replay on the same and fault-free core. If the violation occurs even on fault-free core, then it is a false positive invariant(or a software bug). Since too many rollback/replays due to false positives can cause large overheads, we disable the static invariants once it results in a false positive during dynamic execution (Online updation of invariants will add too much overhead for our software-only technique). In this way, if the total number of static invariants is I, the maximum number of rollbacks possible will also be I, limiting the overhead incurred due to false positives. Currently, the disable operation is done inside the invariant

```
if( (value < min) or (value > max) ){ //Invariant violated
  if( FalsePosArray[InvId] != true ){ //Not false positive
    if( detectFalsePos(InvId) == true )//Perform diagnosis
       FalsePosArray[InvId] = true;//Disable the invariant
} }
```

Figure 1. Invariant checking code template

checking code and does not need any extra hardware support. We maintain a table with one entry for each invariant indexed by the invariant id to identify false positive invariants. In the invariant checking code, the table entry is marked to disable the false positive invariants from all later executions, if the invariant is detected as a false positive. Figure 1 shows a template of the actual invariant checking code.

The overhead caused by false positives in an invariant-based approach depends on the number of rollbacks, which in turn depends on the number of static false positive invariants. The false positive results, presented in Section 5, indicate that the overhead for our set of applications is negligible compared to their runtimes.

4. Methodology

4.1 Simulation Environment

For the fault injection experiments, we used a full system simulation environment comprising the Virtutech Simics full system simulator [31] with the Wisconsin GEMS timing models for the microarchitecutre and the memory [14] as in [9]. These simulators provide cycle-acurate microarchitecture level timing simulation of a real workloads running on a real operating system (full Solaris-9 on the SPARC V9 ISA) on a modern out-of-order superscalar processor and memory hierarchy (Table 1).

We exploit the timing-first approach of the GEMS+Simics infrastructure [15] to inject microarchitecture-level faults. In such an approach, GEMS and Simics compare their full architectural states after each instruction and in case of a mismatch GEMS state is updated with Simics state. This checking mechanism was leveraged for our fault injection. The faults are injected into GEMS's micro-architectural state and the fault is allowed to propagate. After a mismatch between Simics and GEMS, if the mismatch is found to be because of the fault, we copy the faulty architectural state from GEMS to Simics, to make sure Simics follows the same corrupted execution path as GEMS. Otherwise, the GEMS state is updated as usual. When we find the architectural state is corrupted, we say that the fault has been *activated*.

4.2 Fault Model

In the current work, we focus on permanent or hard faults. The well established stuck-at-0 and stuck-at-1 fault models as well as the dominant-0 and dominant-1 bridging fault models are used for modeling permanent hardware faults in this paper. The stuck-at fault models model a fault in a single bit and the bridging fault model models faults that affect adjacent bits. The dominant-0 bridging fault acts like a logical-AND operation between the adjacent bits that are marked faulty, whereas the dominant-1 bridging faults act like a logical-OR operation.

Base Processor Parameters	
Frequency	2.0GHz
Fetch/decode/execute/retire rate	4 per cycle
Functional units	2 Int add/mult, 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 multiply, 24 divide
FP FU latencies	4 default, 7 multiply, 12 divide
Reorder buffer size	128
Register file size	256 integer, 256 FP
Unified Load-Store Queue Size	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

Microarchitecture structure	Fault location
Instruction decoder	Input latch of one of the decoders
Integer ALU	Output latch of one of the integer ALUs
Register bus	Bus on the write port to the register file
Physical integer register file	A physical reg in the integer register file
Reorder buffer (ROB)	Source/dest reg num of instr in ROB entry
Register alias table (RAT)	Logical → phys map of a logical register
Address gen unit (AGEN)	Virtual address generated by the unit
FP ALU	Output latch of one of the FP ALUs

Table 2. Microarchitectural structures in which faults are injected.

The microarchitectural structures and locations where the faults are injected are listed in Table 2. For each structure, a fault is injected in each of 40 random points in each application (after the initialization phase in each application is over). For each application injection point, we perform an injection for each of the 4 fault models (two stuck-at and two bridging faults). The injections are performed on a randomly chosen bit in the given structure. This gives a total of 800 fault injection simulation runs per microarchitectural structure (5 applications × 40 points per application × 4 fault models) and 6,400 total injections across all 8 structures.

After a fault is injected, we run the simulation for 10 million instructions. Note that the fault is maintained for the rest of the 10M instruction window. For the small number of runs where an activated fault is not detected within this window, we use functional (full-system) simulation to run the application to competition (as detailed simulation is too slow to run to completion) for evaluating masking due to the application, and SDCs. The functional simulation does not inject any faults beyond the first 10M instructions, resulting in the fault acting like an intermittent fault that is active only in the 10 million instruction window. We believe that 10M instructions is long enough that the simulation reflects the behavior of permanent faults.

4.3 Fault Detection Techniques Used

We show the effectiveness of our invariant-based approach by evaluating invariants in conjunction with the four low-cost detection mechanisms built into the base SWAT system. This is more realistic than studying only detections by invariants as the other techniques are lower overhead (and need very little hardware/software support) and will show the impact of the new technique in any realistic system.

4.4 Fault Metrics

When the fault causes a corruption in the *architectural state* of the processor, we say it is *activated*. If the fault is never activated, we say the fault is *architecturally masked*. An activated fault which is undetected, but does not cause any corruption in the output produced by the application is said to be *application masked*.

We used five metrics to evaluate the impact of the new detection technique:

1. *Coverage*: The percentage of non-masked faults that are detected in the 10M instruction window. We refer the percentage of undetected faults as *unknown-fraction*.
2. *Latency*: 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.
3. *SDCs*: The number of SDCs which result in corrupting the output of the application.
4. *False positives*: The total number of false positive invariants.
5. *Overhead*: The overhead of the invariant checking code as a percentage of original execution time, measured in fault-free run.

4.5 Applications

For the experiments, we used five SpecCPU 2000 benchmarks – four SpecInt benchmarks (gzip, bzip2, mcf, parser) and one SpecFP benchmark (art). For most of the other SPEC benchmarks, we could not collect sufficient training inputs, while we could not compile and run the others in our simluator.

Previous work on invariants uses toy Siemens benchmarks because many inputs are available for these benchmarks [4, 22]. We use more realistic applications which makes it much harder for us to obtain valid inputs for experiments. Nevertheless, obtaining inputs will not be a problem in practice as developers test their programs on many inputs during the testing phase. Invariant generation and insertion can be easily done during testing through a compile-time pass. The "test" and "train" input sets formed part of our training set. Different techniques were used to generate more inputs depending on the benchmarks. For three benchmarks (gzip, bzip2 and parser), we collected random inputs from external sources. For mcf, a script was used to generate random inputs, while for art, different input options were used to generate invariants. Since the inputs were predominantly generated randomly, the inputs used for training were significantly different from the reference inputs, which we used for testing the false positives, coverage, latency, etc.

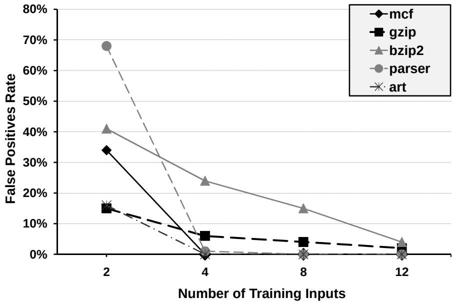

Figure 2. Variation of False positives rate with different number of training inputs. The rate is <5% with 12 training sets, motivating the use of 12 inputs for the rest of our experiments.

5. Experimental Results

In this section, we present our experimental results evaluating the effectiveness of using likely program invariants to detect permanent hardware faults. All the injected FPU faults were architecturally masked in all the applications, except one floating point benchmark (art). So, we have excluded the FPU unit results from the reported results, except as otherwise noted.

We subject the same application binary (instrumented with invariant detection) to faults under both the SWAT and the iSWAT systems. We use the same binary in both cases to obtain a valid coverage comparison between the two cases as the behavior of faults (i.e., whether they are masked, or detected, or become SDCs) depends on both static code layout and dynamic instruction sequence. In the SWAT system, since invariants are not monitored, the system ignores the violation of any invariants and continues execution. The iSWAT framework, on the other hand, invoked the diagnosis module in the case of an invariant to determine false positives. If a false positive is detected, it just continues execution (in this case, the invariant will be disabled in the code), otherwise a fault is detected. [2]

5.1 False Positives

We first evaluate the effect of training with different training sets on the number of false positives.

We define *false positive rate* to be the fraction of false positive invariants as a percentage of total number of static invariants. Figure 2 shows the variation of false positive rate

[2] In a real system, iSWAT should check for false positives on every invariant violation by invoking the rollback/recovery in diagnosis module. However, since we have ref input available, we currently identify the false positive invariants using an offline fault-free run and during faulty run, the diagnosis module uses that information to detect false positives. In this way, we effectively mimic a real system.

for our five applications running on the ref input, as the number of training inputs is increased from 2 to 12.

As expected, false positive rate decreases as the number of inputs increases. By 12 inputs, the rate of false positives is less than 5% for all applications and 0% for three. This false positive rate is sufficiently low for our purpose, motivating us to use 12 training inputs for all of our experiments. In previous work using Siemens benchmarks [4, 22], hundreds of inputs were used for training. We find much lower number of training inputs suffice for permanent fault detection with our approach, as our techniques can tolerate more false positives.

The maximum number of static invariants in all applications was 231. Assuming each false positive detection has an overhead of 1M instructions (conservatively computed considering overheads due to checkpoint/replay and context migration), the maximum overhead of false positive detection on any input will be only 231M instructions, which is negligible considering the application runtimes. In practice, the overhead will even be lower due to low false positive rates.

Interestingly, Figure 2 shows that after just four inputs, only less than 10% of the invariants are false positives for four applications. These results show that likely invariants generated from many inputs will have sufficiently few false positives to be usable for permanent fault detection.

5.2 Coverage

Here, we present the improvements in fault coverage achieved by the iSWAT system (using 12 inputs for training invariants) over the SWAT system, evaluated using micro-architecture-level fault injections.

Table 3 presents the improvements offered by iSWAT over the baseline SWAT system to detect permanent hardware faults. Each column shows the number of fault injections that result in different outcomes (both the absolute number and as a percentage of total number of fault injections) and the last column shows the unknown-fraction. The first two columns represent faults that are masked by the architecture (Arch-Mask), and the application (App-Mask). The Unknown column represents the fraction of faults that are not detected within 10M instructions in each of the systems. The rest of the columns represent faults that are detected by each of the detection mechanisms in 10M instructions, using the detection methods described previously (Section 3). We don't show Abort-App in the table, as it does not detect any fault.

Three points can be observed from this table. First, the invariant detection is catching nearly 5.8% of total fault injections. Second, the invariant detection is detecting some faults that are not detected by the traditional symptoms which resulted in unknowns in SWAT, thus resulting in a **28.6%** reduction of unknown cases from 168 the in SWAT system to 120 in the iSWAT. Third, the iSWAT invariants detect some faults (about 5% of total fault injections) that are caught by the other symptoms in SWAT at a lower latency than the other techniques, thus number of detections by other symptoms in iSWAT are lower compared to SWAT. This result leads to a small improvement in detection latency, as we show in the subsection 5.4. The overall coverage of the iSWAT system is 97.2%.

Microarchitecture structure	SWAT	iSWAT	Reduction
Instruction decoder	0.7%	0.6%	16.7%
Integer ALU	7.8%	6.2%	20.5%
Register bus	4.97%	2.6%	48.3%
Physical integer register file	12.8%	8.5%	33.7%
Reorder buffer (ROB)	0.9%	0.9%	0.0%
Register alias table (RAT)	2.0%	2.2%	-9.6%
Address gen unit (AGEN)	2.4%	1.3%	46.1%
Total	4.0%	2.8%	28.7%

Table 4. Reduction in Unknown category for each microarchitectural structure.

	Unknown	Seg fault	Other signals	No output	SDC
SWAT	168	102	7	28	31
iSWAT	120	85	3	24	8

Table 5. Breakdown of Unknown category after the completion runs. The "No output" category includes OS hangs, application hangs and OS crashes.

Detection using Invariants: In order to understand the effectiveness of these invariants to detect faults in different micro-architectural structures, we categorize the unknowns in the two systems in Table 4. For each structure injected with faults, the table shows the corresponding percentage of non-masked faults that result in unknowns in the SWAT, and iSWAT system, along with the percentage reduction in the unknowns. The "Total" row shows the aggregate numbers.

These results show that invariants are most effective for detecting faults in the integer ALU, register databus, integer register, and AGEN units. These correspond to faults that affect store values, without significantly perturbing the control and data flow. Invariants are not effective for the decode, ROB, and RAT units. Faults in these units perturb program control flow, and do not directly affect values that the invariants monitor. Faults in these structures are also very likely to cause the invariant checking to be done incorrectly. Fortunately, There are a very few remaining unknown cases in these units. Faults in the RAT show an increased unknown rate in the iSWAT system as some faults that are masked the by the application in SWAT, are detected by the invariants in iSWAT. These are real hardware faults which affect program values, but are masked at the application level.

Invariants detect all the unknown cases for FPU Unit faults. Thus the overall unknown-fraction decreases from 4.2% to 2.8%, if we include FPU unit. But, more floating point applications are needed to draw any conclusions.

5.3 SDCs

A small fraction of faults still result in unknown outcomes in the iSWAT system (2.8% of the non-masked faults) after 10M instructions. After 10M instructions of detailed tim-

Symptoms	App-mask	Arch-Mask	Fatal-Trap -App	Fatal-Trap -OS	Hang-App	Hang-OS	INV	High-OS	Unknown	Unknown -fraction
SWAT(%)	293(5.2%)	1090(20%)	1252(22%)	1421(25%)	47(0.8%)	16 (0.3%)	-	1305(23%)	168(3.0%)	(4.0%)
iSWAT(%)	288(5.2%)	1090(20%)	1187(21%)	1357(24%)	29(0.5%)	15(0.3%)	325(5.8%)	1181(21%)	120(2.1%)	(2.8%)

Table 3. Improvement in coverage of iSWAT over SWAT for permanent faults. The percentages are computed using total number of fault injections as the baseline. Invariants are effective in catching faults that escape the traditional detection techniques in SWAT and sometimes catching the same faults earlier, resulting in a reduced 2.8% unknown-fraction compared to 4% of the SWAT system.

ing simulation, we ran the unknown cases (for all structures but the FPU) to completion in functional simulation mode to evaluate how many of the unknown cases result in SDCs. In this mode, faults are not injected during execution due to lack of micro-architectural details in the functional simulator. Also, in functional mode, invariants checks are not enforced by the iSWAT system as we do not have the diagnosis framework support to detect false positives caused by invariants. Hence, our reported SDC numbers are conservative estimates of realistic SDC numbers.

We refer to the number of cases which result in the same output as *App-Mask* and rest of the cases as *unknown*. Table 5 shows the breakdown of the total number of unknown cases according to the results after completion. The next two columns show segmentation faults and application terminations due to other signals. Executions that produce no output due to an application hang, an OS hang or OS Crash (indicated by timing out the execution after a long duration) fall under the *No output* category. Finally, the cases that result in undetected faults that corrupt application outputs are shown under the *SDC* category.

Overall, the SDCs in the iSWAT system is significantly lower than that in SWAT. The invariants reduce the SDCs by **74%**, from **31** to **8**. We consider the reduction in the SDCs as the most important contribution of the invariants. Though a few SDCs remain, we believe that more sophisticated invariants can make the SDC cases negligible. The number of cases detected through other categories also decreases by 27 in iSWAT, which correspond to faults detected by invariants before the application/OS crash through a signal or hang.

Analysis of SDCs: To do an in-depth analysis of why invariants don't detect some of the SDC cases, we moved the invariant checks to the simulator. In this way, we can observe the monitored values and various other information, which is not possible when the checks are in code.

To move the invariant checks into the simulator, we perform an instrumentation pass to store the *monitored invariant values* to known memory locations. The simulator reads the invariants ranges through a file. When it finds a store to the known memory location, it can determine the corresponding invariant from its memory address and perform the bounds checking. Table 6 shows the key results, when the checks are done in the simulator. Overall, there is a 35% reduction in unknown cases and 47% reduction in SDCs. We observe a smaller reduction in SDCs compared to the Table 5. So, the SDC results seem to be sensitive to static code layout and dynamic instruction sequence, as they will determine the instruction where fault is injected and how the fault affects the architectural state.

	Unknown	SDC
SWAT	164	32
iSWAT	106	17

Table 6. Results when invariant checking is done in simulator.

We analyzed the remaining 17 SDC cases by running both the correct runs and fault injection runs and comparing the monitored values. We made some interesting observations:

- In three cases out of 17 (all in mcf), very few invariants are checked after arch-state mismatch. In fact, in one of these cases, the faulty run has much fewer checks compared to correct run, as the control flow diverges completely to some part of the application with much fewer number of checks. In the other two cases, the correct run also has very few invariant checks as the faulty runs.
- In two cases (in mcf,gzip) there are no mismatches of the monitored values inside the 10M window. iSWAT can't detect them as there are no checks after 10M window.
- In most cases, control flow does not diverge significantly - it diverges for a short period and then merges back.
- Irrespective of whether fault injection is in higher or lower order bits, almost all the value mismatches in SDC cases were in lower order bits. So, simple range-based or control-flow invariants are unlikely to be effective for these cases. In fact, most of the value mismatches were in lowest 3 bits. We will need other types of invariants/detectors for detecting mismatches in lower order bits.

5.4 Latency

Table 7 shows latency results for the faults detected by the SWAT and the iSWAT systems, binned into various categories from under 1k instructions to under 10M instructions. In order to perform a fair comparison, the numbers are presented as a percentage of the total number of faults detected by iSWAT (i.e. the number of detection in the <10M case).

The number of faults detected at a latency of under 1k instructions shows the largest increase of about 2% (the rest of the numbers are cumulative). This shows that the latency of detection of invariants is significantly lower than that of the other symptoms. This increases the number of faults that are amenable to simple hardware recovery. Although the latency benefits offered by iSWAT are not substantial

Latencies	<1k	<5k	<10k	<50k	<100k	<500k	<1M	<5M	<10M
SWAT	41.1%	47.0%	50.7%	78.7%	81.0%	87.0%	90.3%	95.7%	98.7%
iSWAT	43.1%	49.6%	53.4%	81.2%	83.3%	89.2%	92.7%	97.7%	100.0%

Table 7. Detection latencies for SWAT and iSWAT. The percentages are computed using number of detections in iSWAT with <10M as baseline. The invariants increase the faults amenable to hardware recovery by 2%.

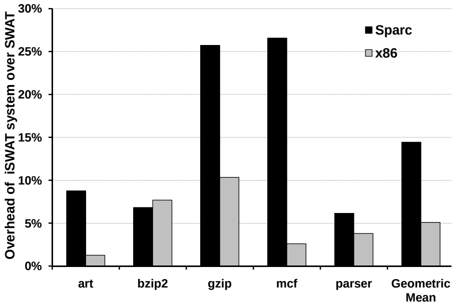

Figure 3. Overhead of invariants on an UltraSPARC-IIIi (sparc) machine and an AMD Athlon machine (x86).

so far, using more sophisticated invariants may improve the effectiveness of iSWAT to reduce the latency.

5.5 Overhead

We evaluate the overhead of using invariants by running the binary (with invariants checking) on fault-free hardware, using two machines: Sun UltraSPARC-IIIi 1.2GHz machine with 1MB unified L2, and 2GB RAM, and on an AMD Athlon(TM) dual-core MP 2100+ machine with 256KB L2 and 1.5GB RAM. The Sun machine is referred to as *Sparc* machine, and the AMD one as *x86* machine in this section.

Figure 3 shows the overhead of using invariants checking in the programs as a percentage over the baseline program which has no invariants checking. The geometric mean of the overheads is also shown for the two machines.

The sparc machine exhibits a higher overhead when running the invariants code than the x86 machine, with the average overheads being 14% and 5% respectively. In particular, the overhead for the application mcf is significantly higher in the Sparc machine (26%) than the x86 machine (2%). The high overhead of the Sparc machine is likely due to its inability to hide the cache misses and branch mispredictions induced by these extra invariants. The x86 machine is able to hide these latencies better, resulting in lower overheads.

In spite of these differences, the overheads produced by these invariants checks are within acceptable overheads for the increased coverage that they provide, motivating the iSWAT system for increased resilience.

6. Related Work

There is a growing body of work on using *software-visible symptoms* to detect hardware errors. A number of papers propose the use of control path signatures to detect control-flow errors [1, 5, 21, 29, 30]. Wang and Patel propose using branch mispredictions, cache misses, and exceptions as symptoms of faults [32]. Most of this work focuses on transient faults or intermittent faults, and does not handle permanent faults (the exceptions are discussed below). Permanent faults are important because of the expected increase in phenomena such as wear-out, insufficient burn-in, and design defects [2]. In our previous work [9], we used simple software symptoms to detect both permanent as well as transient faults and we extend that system in this paper.

Dynamically detected program invariants (likely invariants [4]), which are inherently unsound, have been studied for a wide range of *analysis* tasks, including program evolution [4], program understanding [7, 4], and detecting and diagnosing software bugs [4, 6, 12, 33, 13]. The only work we know of that uses likely invariants for *online* error detection comprises three recent papers on transient hardware fault detection [22, 24, 3]. Racunas et al. and Dimitrov et al. extract the invariants using online hardware monitoring whereas Pattabiraman et al. use ahead-of-time monitoring of program runs (similar to our work). In all the cases, however, they can only use their invariants for transient errors because they do not have any mechanism to distinguish false positives from true hardware failures. (Racunas et al. and Dimitrov et al. flush the pipeline, Pattabiraman et al. do not suggest any concrete solution for false positives.) In contrast, we can handle both transient and permanent faults.

Meixner et al. in the Argus project have proposed the use of a program dataflow checker, combined with control flow signature checking, functional unit checkers, a memory checker and parity on all data transfer and storage units to handle a wide range of faults [16, 17]. Their dataflow graph and control flow signatures conceptually are invariants that are encoded by the compiler in the binary and checked by the hardware. Unfortunately, the technique does not work with interrupts, I/O, etc. because these affect the control flow. Some parts of the Argus solution may also incur inordinate performance overhead. Coverage data is reported only for a synthetic microbenchmark, thus the effectiveness of the technique for real programs is not clear [16]. Finally, the estimated area overhead is 17% of the core for Argus – a fault in this part could lead to false positives. In contrast, we look at far cheaper detection techniques, combining software-extracted invariants with several other software symptoms that can be observed at near-zero cost.

7. Conclusion and Future Work

Previously existing methods for detecting hardware faults using software-level symptoms, such as SWAT [9], are very promising because of their high coverage and low cost. Nevertheless, these systems need additional detectors for achieving reliability levels that would be acceptable for most systems. In this work, we proposed and evaluated the first design (we know of) that uses likely program invariants for detecting permanent faults. We used simple range-based invariants on single variable values, in conjunction with low-overhead symptom-based detection techniques already available in the SWAT System. Our results show that likely invariants can reduce the fraction of undetected errors from **4%** to **2.8%**, when used in conjunction with other symptom-based detection techniques. Further, they reduce SDCs by **47%** to **74.2%**, which is important for any hardware fault tolerance solution. We further showed that by leveraging the diagnosis framework in SWAT, we could keep the overhead caused by false positives to acceptable levels.

These range-based invariants form a first step towards using invariants to detect hardware faults. We are now investigating more sophisticated invariant schemes to further improve the effectiveness of the iSWAT system. We also want to monitor other program values and to design a strategy to select the most effective values for monitoring to reduce overhead. We would also like to evaluate the approach on more benchmarks and real applications.

Acknowledgments

We would like to thank Robert Bocchino for many discussions and help in writing.

References

[1] E. Borin, C. Wang, Y. Wu, and G. Araujo. Dynamic binary control-flow errors detection. *SIGARCH Comput. Archit. News*, 33(5), 2005.

[2] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6), 2005.

[3] M. Dimitrov and H. Zhou. Unified architectural support for soft-error protection or software bug detection. In *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[4] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 2001.

[5] O. Goloubeva et al. Soft-Error Detection Using Control Flow Assertions. In *Proc. of 18th IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI Systems*, 2003.

[6] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, 2002.

[7] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *IEEE Int'l Conf. on Software Maintenance (ICSM)*, 2001.

[8] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. Int'l Symp. on Code Generation and Optimization*, 2004.

[9] 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 *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2008.

[10] M. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Trace Based Diagnosis of Permanent Hardware Faults. In *International Conference on Dependable Systems and Networks*, 2008.

[11] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proc. of Conf. on Programming Language Design and Implementation*, 2003.

[12] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. of Conf. on Programming Language Design and Implementation*, 2005.

[13] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

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

[15] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-System Timing-First Simulation. *SIGMETRICS Performance Evaluation Rev.*, 30(1), 2002.

[16] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proc. ACM/IEEE Int'l Symposium on Microarchitecture*, 2007.

[17] A. Meixner and D. J. Sorin. Error detection using dynamic dataflow verification. In *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[18] J. Nakano et al. ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In *Int'l Symp. on High Performance Computer Architecture (HPCA)*, 2006.

[19] N. Nakka et al. An Architectural Framework for Detecting Process Hangs/Crashes. In *European Dependable Computing Conference (EDCC)*, 2005.

[20] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proc. ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, 2002.

[21] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *IEEE Trans. on Reliability*, 51, March 2002.

[22] K. Pattabiraman, G. P. Saggesse, D. Chen, Z. Kalbarczyk, and R. Iyer. Dynamic derivation of application-specific error detectors and their hardware implementation. In *Proc. of European Dependable Computing Conference (EDCC)*, 2006.

[23] M. Prvulovic et al. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Int'l Symp. on Computer Architecture (ISCA)*, 2002.

[24] P. Racunas et al. Perturbation-based Fault Screening. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[25] V. Reddy et al. Assertion-Based Microarchitecture Design for Improved Fault Tolerance. In *International Conference on Computer Design* , 2006.

[26] G. A. Reis et al. Software-Controlled Fault Tolerance. *ACM Transactions on Architectural Code Optimization*, 2(4), 2005.

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

[28] D. Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Int'l Symp. on Computer Architecture (ISCA)*, 2002.

[29] R. Vemu and J. A. Abraham. CEDA: Control-flow Error Detection through Assertions. In *Intl. On-Line Test Symposium*, 2006.

[30] R. Venkatasubramanian et al. Low-Cost On-Line Fault Detection Using Control Flow Assertions. In *International On-Line Test Symposium*, 2003.

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

[32] 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.

[33] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proc. ACM/IEEE Int'l Symposium on Microarchitecture*, 2004.