Automated Derivation of Application-specific Error Detectors Using Dynamic Analysis

Karthik Pattabiraman (Member, IEEE), Giacinto Paolo Saggese (Member, IEEE), Daniel Chen (Member, IEEE), Zbigniew Kalbarczyk (Member, IEEE), and Ravishakar K. Iyer (IEEE Fellow)

Abstract: This paper proposes a novel technique for preventing a wide range of data errors from corrupting the execution of applications. The proposed technique enables automated derivation of fine-grained, application-specific error detectors based on dynamic traces of application execution. The technique derives a set of error detectors using rule-based templates to maximize the error detection coverage for the application. A probability model is developed to guide the choice of the templates and their parameters for error-detection. The paper also presents an automatic framework for synthesizing the set of detectors in hardware to enable low-overhead, run-time checking of the application. The coverage of the derived detectors is evaluated using fault injection experiments, while the performance and area overhead of the detectors is evaluated by synthesizing them on reconfigurable hardware.

Keywords— Data Errors, Dynamic Execution, Likely Invariants, Critical Variables, FPGA Hardware

1 INTRODUCTION

THIS paper presents a technique to derive and implement error detectors that protect programs from data errors. These are errors that cause a divergence in data values from those in an error-free execution of the program. Data errors can cause the program to crash, hang, or produce incorrect output (fail-silent violations). Such errors can result from incorrect computation, and they would not be detected by generic techniques such as Error Correcting Codes (ECC) in memory and/or registers.

Many static and dynamic analysis techniques [1-4] have been proposed to find bugs in programs. However, these techniques are not geared toward detecting runtime errors, as they do not consider error propagation. To detect runtime errors, we need mechanisms that can provide high-coverage, low-latency (rapid) error detection to: (i) preempt uncontrolled system crash/hang and (ii) prevent propagation of erroneous data and limit the extent of the (potential) damage. Eliminating error propagation is essential because programs, upon encountering an error that could eventually lead to a crash, may execute for billions of cycles before crashing [5]. During this time, the program can exhibit unpredictable behavior, such as writing corrupted state to a checkpoint or sending a corrupted message to another process [6], which in turn can result in extended downtimes [7].

It is common practice for developers to write assertions in programs for detecting runtime errors [8]. However, assertions are often cumbersome to write, and they require considerable programmer effort and expertise to develop correctly. Further, placing an assertion in the wrong place can hinder its detection capabilities [9]. As a result, programmer-written assertions (by themselves) are not very effective in providing high coverage of runtime errors [10].

Hiller et al. propose a technique to derive assertions in an embedded application based on the high-level behavior of its signals [11]. They facilitate the insertion of assertions by means of well-defined classes of signal patterns. In a companion paper, they also describe how to place assertions by performing extensive fault-injection experiments [12]. However, this technique requires extensive knowledge of the application. Further, performing fault-injection may be time-consuming and cumbersome for the developer. Therefore, it is desirable to develop an automated technique to derive and place detectors in application code.

Our goal is to devise detectors that preemptively detect errors impacting the application and to do so in an automated way without requiring programmer intervention or fault-injection into the system. In this paper, the term "detectors" refers to executable assertions used to detect runtime errors. The main contributions are as follows:

1. Derivation of error detectors based on the dynamic execution traces of the application instrumented at strategic locations in the code;
2. Introduction of rule-based templates that capture common patterns of the temporal behavior of variables in an application;
3. Development of a probability model for estimating detection coverage that can be used to guide the derivation process to maximize detection coverage for a

- *Karthik Pattabiraman is with the University of British Columbia, Canada. E-mail: karthikp@ece.ubc.ca*
- *Giacinto Paolo Sagesse is with Synopsys Inc. Email: saggese@gmail.com*
- *Daniel Chen is with the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign (UIUC). Email: dchen8@uiuc.edu*
- *Zbigniew Kalbarczyk is with the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign (UIUC). Email: kalbarcz@uiuc.edu*
- *Ravishankar Iyer is with the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign (UIUC). Email: rkiyer@uiuc.edu*

given cost; and
4. Synthesis of custom hardware (VHDL code) to implement the derived detectors on Field Programmable Gate Arrays (FPGAs) so that they can be executed in parallel with the application.

The proposed methodology is applied to derive a set of detectors for several benchmark programs. Experimental evaluation of coverage is performed via random fault injection into application data when the application is executed on a hardware simulator. We also synthesize the detectors on FPGA hardware to measure the additional hardware resources and performance overheads incurred by the detectors. This paper builds upon the work in our earlier conference paper on this topic by comparing the coverage of the derived detectors with the highest coverage that can be obtained using detectors based on single data values (best-value detectors) [13]. The paper also compares the proposed technique with other techniques that have been published since our conference paper.

The main results obtained from the study are:
1. The derived detectors detect 50-75% of errors that manifest at the application level, when 100 detectors are placed in the application code (this corresponds to about 5% of a program's assembly code, not including libraries);
2. Hardware implementation of the detectors results in a performance overhead of about 5%, with area and power overheads of about 15% and 5%, respectively;
3. False positives (detectors flag an error when no error is present) are less than 2.5% when the training data constitutes 20% of the input sets used for testing; and
4. The coverage achieved by the derived detectors is close to the coverage provided by the best-value detectors (70 to 90% of manifested faults).

2 Approach and Fault Model

The derivation and implementation of the error detectors in hardware and software encompasses four main phases, as shown in Figure 1.

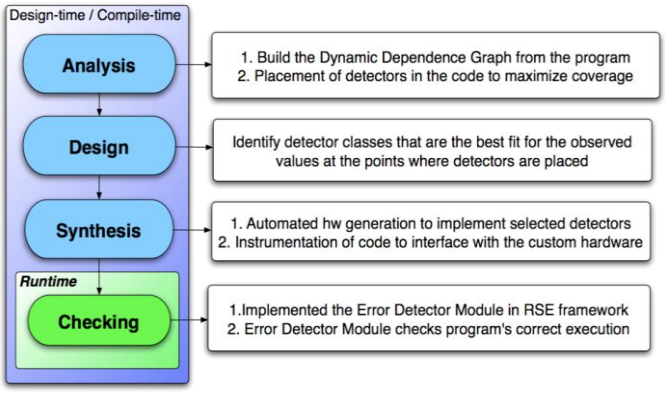

Figure 1: Steps in detector derivation process

The *analysis phase* identifies the program locations and variables for detector placement based on the Dynamic Dependence Graph (DDG) of the program. Fault-injections are not required. We choose the locations for detector placement using the *Fanouts* heuristic [14]. The program code is then instrumented to record the values of the chosen variables at the locations selected for detector placement.

The *design phase* uses the dynamic traces of recorded values over multiple executions of the application in order to choose the best detector that matches the observed values for the variable, based on a set of predetermined generic detector classes (Section 3). The best detector is the one that maximizes detection coverage with the lowest cost. This is defined in terms of a probability model.

After this stage, the detectors can either be integrated into application code as software assertions or implemented in hardware. In this paper, we consider a hardware implementation of the derived detectors in order to minimize detection latency and performance overheads.

The *synthesis phase* converts the generated assertions to a Hardware Description Language (HDL) representation that is synthesized in hardware. It also inserts special instructions in the application code to invoke and configure the hardware detectors. This is explained in Section 4.

Finally, during the *checking* phase, the custom hardware detectors are deployed in the system to provide low-overhead, concurrent run-time error detection for the application. When a detector detects a deviation from the application's behavior learned during the design phase, it flags an error and halts the program.

Note that the analysis and design phases are related to the derivation of the detectors, while the synthesis and checking phases are related to the implementation and deployment of the derived detectors, respectively.

Fault Model. The fault model covers errors in the data values used in the program's execution. This includes faults in: (1) the *instruction stream*, which result in the wrong op-code being executed or the wrong registers being read or written by the instruction, (2) the *functional units of the processor*, which can result in incorrect computations, (3) the *instruction fetch and decode units*, which can result in an incorrect instruction being fetched or decoded, and (4) the *memory and data bus*, which can cause wrong values to be fetched or written in memory and/or processor register file. Note that these errors would not be detected by techniques such as ECC (Error Correcting Codes) in memory, as they originate in the computation.

It is important to understand that some of the faults mentioned above will fail to result in data value errors, in which case they will not be detected by the proposed technique. However, studies have shown that 60-70% of errors in the processor result in data value errors [15-16].

The fault-model also represents certain types of software errors that result in data-value corruptions, such as: (1) synchronization errors or race conditions, which can result in corruptions of data values due to incorrect sequencing of operations, (2) memory corruption errors, e.g., buffer-overflows and dangling pointer references, which can cause arbitrary data values to be overwritten in

memory, and (3) the use of uninitialized or incorrectly initialized values, which can result in the use of unpredictable values depending on the platform and environment. These errors often escape conventional testing and debugging techniques and manifest themselves in the field [17].

3 Derivation: Analysis and Design

In this paper, an *error detector* is an assertion based on the value of a single variable[1] of the program at a specific location in its code. A detector for a variable is placed immediately *after* the instruction that writes to the variable. Since a detector is placed in the code, it is invoked each time the detector's location is executed. An example code fragment is shown in Table 1.

Table 1: Example code fragment

```
void foo() {
    int k = 0;
    for (; k<N; k++) {
    ….
    }
}
```

In the example, assume that the detector placement methodology has identified variable k as the critical variable to be checked within the loop. Although this example illustrates a simple loop, our technique is general and does not depend on the structure of the source program (or even require the program's source code for the matter). Nevertheless, we show the source-level representation of the code for clarity.

Variable k is initialized at the beginning of the loop and incremented by 1 within the loop. Within the loop, the value of k depends on its value in the previous iteration. Hence, the rule for k can be written as "either the current value of k is zero, or it is greater than the previous value of k by 1." We refer to the current value of the detector variable k as k_i and the previous value as k_{i-1}. Thus, the detector can be expressed as: $(k_i - k_{i-1} == 1)$ *or* $(k_i == 0)$.

As seen from the example, a detector can be constructed for a target variable by observing the dynamic evolution of the variable over time. The detector consists of a rule describing the allowed values of the variable at the selected location in the program, and an exception condition to cover correct values that do not fall into the rule. If the detector rule fails, then the exception condition is checked, and if this also fails, the detector flags an error. Detector rules can belong to one of six generic classes and are parameterized for the variable checked. The rule classes are shown in Table 2. These rule classes are chosen based on observations about the behavior of program variables and are similar to those in Hiller et al. [11].

The exception condition involves equality constraints on the current and previous values of the variable, as well as logical combinations (*and*, *or*) of two of these constraints. The equality constraints take the following forms: $a_i == d$, where d is a constant parameter; $a_{i-1} == d$, where d is a constant parameter; and $a_i == a_{i-1}$. However, not all combinations of the above three clauses are logically consistent. For example, the exception condition ($a_i == 1$ *and* $a_i == 2$) is logically inconsistent, as a_i cannot take two different values at the same time. Of the 27 possible combinations of the clauses, only 8 are logically consistent. Hence, there are a total of 48 rule class, exception pairs that can be used to construct a detector for a particular location.

For the example involving the loop index variable k, discussed in this section, the rule class is *Constant-Difference* of 1, and the exception condition is $(k_i == 0)$.

3.1 Dynamic Derivation of Detectors

This section describes our overall methodology for automatically deriving detectors based on the dynamic trace of values produced during the application's execution. By *automatic derivation*, we mean determination of the rule and the exception condition for each variable targeted for error detection. The basic steps in the algorithm are:

1. The program points for the placement of detectors (both variables and locations) are chosen based on the Dynamic Dependance Graph (DDG) of the program as shown in prior work[14].
2. The program is instrumented to record the run-time evolution of the values of the detector variables at their respective locations and executed over multiple inputs. This obtains dynamic traces of the checked values.
3. The dynamic traces of the values obtained are analyzed to choose a set of detectors (both rule class and exception condition) that match the observed values.
4. A probabilistic model is applied to the set of chosen detectors to find the best detector for a given location. The best detector is characterized in terms of its tightness and execution cost (see next section).

3.2 Detector Tightness and Execution Cost

A qualitative notion of the *tightness* of a detector was first introduced in [18]. We define tightness in a quantitative sense as the conditional probability that the detector detects an error, given that there is an error in the value of the variable that it checks. Note that this is not the same as the coverage, as the *coverage* of the detector is the probability that it detects an error given that there is an error in *any* value used in the program. Hence, in addition to tightness, coverage also depends on the probability that an error propagates to the detector's variable.

To characterize the tightness of a detector, we must consider both the rule and the exception condition, as the error will not be detected if either passes. Tightness also depends on the parameters of the detector and the probability distribution of the observed stream of data values in a fault-free execution of the program. For an incorrect value to go undetected, either the rule or the exception condition or both must evaluate to true. This can happen in four mutually exclusive ways, as shown in Table 3.

[1] In this paper, a *variable* refers to any register, cache or memory location.

Table 2: Generic rule classes

Class Name	Generic Rule (a_i , a_{i-1})	Description
Constant	$(a_i == c)$	The value of the variable in the current invocation of the detector is a constant given by parameter *c*.
Alternate	$((a_i == x \,/\backslash\, a_{i-1} == y)) \,\backslash/\, (a_i == y \,/\backslash\, a_{i-1} == x)$	The values of the variable in the current and previous invocations of the detector alternate between parameters *x* and *y*, respectively.
Constant-Difference	$(a_i - a_{i-1} == c)$	The value of the variable in the current invocation of the detector differs from its value in the previous invocation by a constant *c*.
Bounded-Difference	$(min <= a_i - a_{i-1} <= max)$	The difference between the values of the variable in the previous and current invocations of the detector lies between *min* and *max*.
Multi-Value	$a_i \epsilon \{ x, y, \ldots \}$	The value of the variable in the current invocation of the detector is one of the set of values *x*, *y*, …
Bounded-Range	$(min <= a_i <= max)$	The value of the variable in the current invocation of the detector lies between the parameters *min* and *max*.

Table 3: Probability values for computing tightness

Symbol	Explanation
$P(R \mid R)$	Probability that an error in a value that originally satisfied the rule (in a correct execution) also causes the incorrect value to satisfy the rule.
$P(R \mid X)$	Probability that an error in a value that originally satisfied the exception condition (in a correct execution) causes the incorrect value to satisfy the rule.
$P(X \mid R)$	Probability that an error in a value that originally satisfied the rule (in a correct execution) causes the incorrect value to satisfy the exception condition.
$P(X \mid X)$	Probability that an error in a value that originally satisfied the exception condition (in a correct execution) causes the incorrect value to satisfy the exception condition.

The tightness of a detector is defined as *(1 – P(I))*, where P(I) is the probability of an incorrect value passing undetected through the detector. This probability can be expressed using the terms in Table 3, as follows:

$$P(I) = P(R) [P(R \mid R) + P(X \mid R)] + P(X) [P(R \mid X) + P(X \mid X)] \quad (1)$$

where *P(R)* is the probability of the value belonging to the rule, and *P(X)* is the probability of the value belonging to the exception condition, both of which are derived from the observed value stream on a per-application basis.

The computation of tightness can be automated, since there are only a limited number of rule-exception pairs[2]. These probabilities can be precomputed as a function of the detector's parameters as well as on the frequency of elements in the observed data stream for each rule-exception pair. We do not list all the probabilities, but instead illustrate with an example.

Example of Tightness Calculation. Consider a detector in which the rule expression belongs to the class *Bounded-Range* with parameters *min = 5* and *max = 100,* and the exception condition is of the form $(a_i==0)$. We make the following assumptions about errors in the program:

1. The distribution of errors in the detector variable is uniform across the range of all possible values the variable can take for its type (say, *N*);
2. An error in the current value of the variable is not affected by an error in the previous value of the variable; and
3. Errors in one detector location are independent of errors in another detector location.

These are optimistic assumptions, hence the estimation of tightness is an upper bound on the actual value of the detector's tightness (and hence its coverage). Relaxing these assumptions may yield higher accuracy, but it requires apriori knowledge of the application's semantics and error behavior in the application, which cannot be obtained through dynamic analysis.

Table 4 shows the precomputed probability values for this detector in terms of *N* and the detector's parameters.

Table 4: Probability values for computing tightness of detector "Bounded-Range (5, 100) except: $(a_i==0)$"

Symbol	Value	Explanation
$P (R \mid R)$	$(95 / N)$	Each rule value can turn into any of the other 95 rule values with equal probability.
$P (R \mid X)$	$(96 / N)$	An exception value can turn into one of 96 rule values with equal probability
$P (X \mid R)$	$(1 / N)$	A rule value can incorrectly satisfy the exception condition if it turns into 0.
$P (X \mid X)$	0	An exception value cannot change into another exception value, as there is only one value permitted by the exception condition (in this example, the value is 0).

Substituting these probability values in equation (1):

$$P(I) = P(R) [95/N + 1/N] + P(X) [96/N + 0] = (96/N)[P(R) + P(X)] = \mathbf{96/N}$$

The above derivation uses the fact that $P(R) + P(X) = 1$, since the value must satisfy either the rule or the exception in an error-free execution, and since the two events are mutually exclusive.

Consider a new detector in which the rule belongs to the *Constant* class (with parameter 5). Let us assume that the exception condition is the same as the old detector's. For this new detector:

$$P(R \mid R) = 0,\ P(R \mid X) = 1/N,$$
$$P(X \mid X) = 0 \text{ and } P(X \mid R) = 1/N$$

Substitution of the above values in equation (1), yields the following expression for *P(I):*

[2] There are 6 types of rule classes and 8 types of exception conditions, leading to a total of 48 rule-exception pairs.

$P(I) = P(R)\,[\,0 + 1/N\,] + P(X)\,[1/N + 0\,]$
$= (1/N)[\,P(R) + P(X)\,] = \mathbf{1/N}$

Note that the probability of missing an error in the first detector is 96 times the probability of missing an error in the second detector. The tightness of the first detector is correspondingly much less than the tightness of the second (which matches with our intuition).

The above model is used only to compare the relative tightness of the detectors, not to compute the actual probability values, which may be very small. The range of values for the detector variable represented by the symbol *N* gets eliminated in the comparison among detectors for the same variable/location, and it does not influence the choice of the detector.

Execution Cost. The execution cost of a detector is the amortized computation cost in executing the detector over multiple values observed at the detector point. The execution cost of a detector is calculated as the number of basic arithmetic and comparison operations executed in a single invocation of the detector, averaged over the entire lifetime of the program's execution. An operation usually corresponds to a single arithmetic or logical operator. Note that the computation of the execution cost assumes an error-free execution of the program.

For example, the detector considered above has two comparison operations for the rule and one comparison operation for the exception. Assume that the rule is satisfied 80% of the time, which implies that the exception condition is satisfied the remaining 20% of the time, i.e., 80% of the data points in the trace satisfy the rule while 20% satisfy the exception condition. Therefore, the total execution cost for the detector is (2 * 0.8 + 3 * 0.2 =) 2.2 operations[3].

3.3 Detector Derivation Algorithm

For each location identified by the detector placement analysis in [14], the detector derivation algorithm first chooses the rule class corresponding to the detector location and then forms the associated exception condition. The algorithm attempts to maximize the tightness to execution cost ratio for the detector. We refer to the evolution of a program variable over time as the *stream of values* for that variable. The steps in the algorithm are as follows:

1. To derive the rule, each of the rule classes in Table 2 is tried in sequence against the observed value stream to determine which rule classes satisfy the observed value stream. The parameters of the rule are learned from appropriate samples (for each rule class) from the observed stream. For the same location, it is possible to generate multiple rules that are considered candidates for exception derivation in the next step.
2. For each rule derived in step 1, the associated exception condition is derived. Each value in the stream that does not satisfy the rule is used as a seed for generating exception conditions for that rule (through exhaustive search among the exception conditions). If it is not possible to derive an exception condition for the observed value according to the conditions in Section 3, the current rule is discarded and the next rule is tried from the set of rules derived in step 2.
3. For each rule-exception pair generated, the tightness and execution cost of the detector are calculated. The detector with the highest tightness to execution cost ratio is chosen as the final detector for that location and exported to a text file for synthesis to hardware.

The time complexity of the above algorithm is directly dependent on the number of values observed at each detector location (say *m*), the number of detector locations considered in the application (say *n*), and the number of streams or inputs on which the algorithm is trained (say *k*). The time also depends on the number of rules and exception classes, both of which are constants. Therefore, the overall time complexity of the algorithm is given by $O(m * n * k)$.

4 Deployment: Synthesis and Hardware Implementation

This section discusses the hardware implementation of the detectors derived using the algorithm in Section 5, and it can be skipped if the reader is not interested in the hardware details.

In this paper, we discuss the hardware implementation of the derived error detectors in the context of the Reliability and Security Engine (RSE) framework [19]. The RSE is a reconfigurable, processor-level framework that provides reliability and security functions according to the requirements and characteristics of the application. The RSE Framework consists of *RSE modules*, which provide the reliability and security services, and the *RSE Interface,* which provides a standard, well-defined and extensible interface between the modules and the main processor pipeline. The interface collects the intermediate pipeline signals and converts it to a generic format that can be used by the RSE modules for error and attack detection. The application interfaces with the RSE modules using special instructions called CHECK instructions.

In this paper, we consider a simple DLX processor [20] augmented with the RSE. The DLX is a RISC processor with a five-stage pipeline with in-order issue but out-of-order execution. The detectors are implemented as a separate module of the RSE called the *Error Detector Module* (EDM). The detectors are configured into the EDM at application-load time and are invoked from the application using CHECK instructions.

4.1 Synthesis of Error Detector Module

The output of the detector derivation algorithm in Section 3.3 is a list of detectors in the program. This list is used to synthesize hardware checkers that implement the derived detectors in the EDM. The application is instrumented with CHECK instructions to invoke the hardware checkers, and the EDM is generated using the synthesized hardware checkers. These two steps are carried out at compile time through an automated design flow, illustrated in Figure 2.

[3]When a detector is invoked, the rule is checked first, and only if it fails is the exception condition checked for the value.

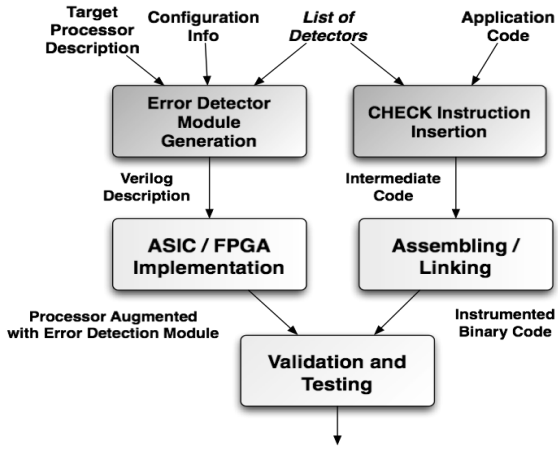

Figure 2: Design flow to instrument application and generate the EDM from the list of detectors

Figure 2 shows the automated design flow from the application code to the hardware. Given the application code (in the form of assembly code), the design flow produces the instrumented application code and the hardware description of the EDM tailored for the target application. The *target processor description* (a DLX-like processor in the current implementation) and the *configuration information* are provided as parameters to the design flow. These are used to extract, from the main pipeline of the processor, the signals needed by the EDM for performing error detection. The output of the *Error Detector Module generation* phase in Figure 2 is a Virtual Hardware Description Language (VHDL) representation, which is in turn used by the synthesis phase to instantiate the various hardware components considered in Section 6.

Each detector in the *list of detectors* derived in the design phases is characterized by the following attributes: (1) location of the detector in terms of the Program Counter (PC) value at which it is to be invoked, (2) processor registers that must be checked by the detector, and (3) detector class and exception parameters. Figure 3 shows the format of each detector, which consists of six words.

PC	Rule Class				Exception				
	Class	Register	Param 1	Param 2	Operator	Class 1	Class 2	Param 1	Param 2
32 bit	3 bit	5 bit	32 bit	32 bit	2 bit	2 bit	2 bit	32 bit	32 bit

Figure 3: Format of detector and bit width

In our current deployment, the *application* is represented as assembly code. The header of the assembly file is instrumented with special instructions to load the detectors of the application into the EDM. Each of these instructions loads a single 32-bit value, and since each detector consists of six words, we need six instructions per detector to perform the loading. However, these instructions are executed only once during the lifetime of the application. The application code is also instrumented with CHECK instructions to invoke the detectors during its execution.

4.2 Structure of Error Detector Module

Figure 4 shows the overall architecture of the Error Detector Module (EDM). As mentioned before, the EDM is implemented as a module in the Reliability and Security Engine (RSE). As shown in the figure, the RSE interface extracts signals of interest from the processor's pipeline and conveys this information to the EDM for use in detection. The main components of the EDM are as follows:

The *Shadow Register File* (SRF) keeps track of current and last values of the microprocessor's registers checked by the detectors (i.e., a_i and a_{i-1}, where a can be any architectural register). This component delivers the required values a_i and a_{i-1} when a detector is executed (based on the detector's rule and exception condition). When a new value *regValue* is written at time i by the processor in register R of the processor file (based on the value *regSel*), a copy of the new value R_i is stored in the SRF. The old value R_{i-1} is also retained. Since not all the registers of the processor architecture have to be checked by the detectors, a mapping between the physical addresses of the microprocessor registers and the logical addresses of the corresponding registers in the SRF is kept in a hardware structure named *Phys2Log*.

The *Detector Table* stores the information needed to execute a detector. The size of the table is directly proportional to the number of detectors needed by an application. It consists of comparators for checking the current PC against the PCs of the detectors and triggering them if necessary and a Random Access Memory (RAM) for storing the parameters of rules and exceptions. When a detector is triggered by the current PC, the Detector Table first selects the register R that has to be checked from the SRF, which in turn forces the values R_{i-1} and R_{i-1} to be placed on the dual data-path buses. It then activates the Rule and Exception Checkers to perform the computations associated with the detector. If the computations fail, the Error Signal Computation flags the Violation Detection signal.

Rule and Exception Checkers are the actual data-paths used to carry out the computation of the detector rules and exception conditions. A number of checker components are instantiated to perform the required computations according to the rule classes and exceptions needed by an application. Note that the number of checkers instantiated is equal to the number of detector classes and exceptions (at most 48) rather than to the number of detectors inserted in an application.

5 Experimental Evaluation

This section describes the experimental infrastructure and application workload used to evaluate the coverage and overheads of the derived detectors. We use fault-injection on the application executed to completion in a processor simulator to evaluate the coverage of the derived detectors. We implement the detectors on Field Programmable Gate Array (FPGA) hardware to evaluate their performance and area overheads.

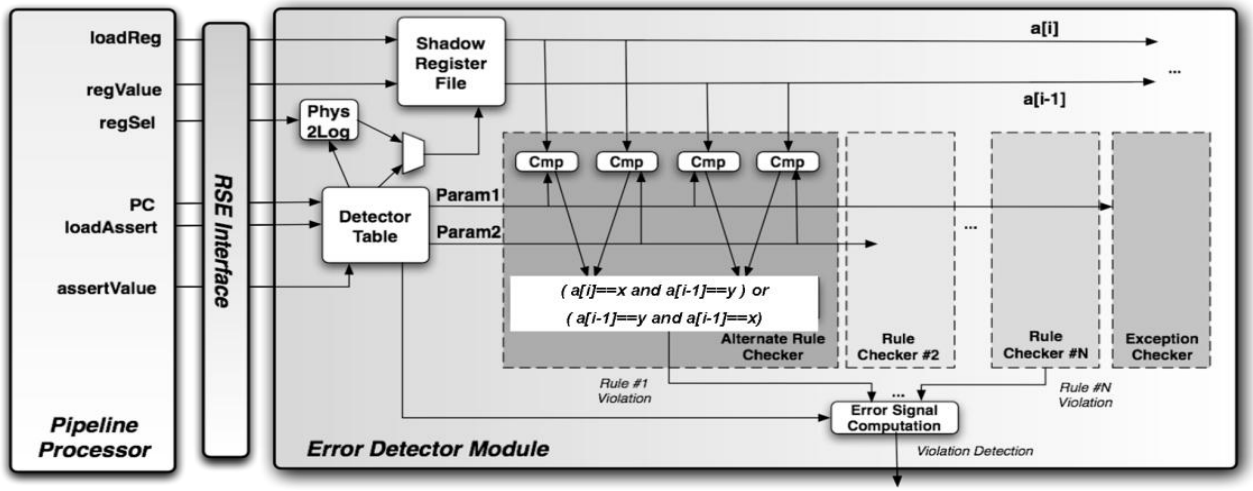

Figure 4: Architectural diagram of synthesized processor

5.1 Application Programs

The system is evaluated with six of seven programs from the Siemens suite [21] of programs (Table 5). These programs are equipped with extensive test suites.

Table 5: Benchmark Programs

Benchmark	Description
Replace	Searches a text file for a regular expression and replaces the expression with a string
Schedule, Schedule2	A priority scheduler for multiple job tasks
Print_tokens, Print_tokens2	Breaks the input stream into a series of lexical tokens according to pre-specified rules
Tot_info	Offers a series of data analysis functions

5.2 Infrastructure

The tracings of the application's execution and the fault-injections are performed using a functional simulator in the *SimpleScalar* family of processor simulators [22]. The simulator allows fine-grained tracing of the application without modifying the application code and provides a virtual sandbox in which to execute the application and study its behavior under faults.

We modified the simulator to track dependencies among data values in both registers and memory by shadowing each register/location with four extra bytes (invisible to the application) that store a unique tag for that location. For each instruction executed by the application, the simulator prints (to the trace file) the tag of the instruction's operands and the tag of the resulting value.
The trace file is analyzed offline by specialized scripts to construct the DDG and compute the metrics for placing detectors in the code as described in our prior work [14].

The effectiveness of the detectors is assessed using fault injection. Fault locations are specified randomly from the dynamic set of tags produced in the program. In this mode, the tags are tracked by the simulator, but the executed instructions are not written to the trace file. When the current instruction's tag value equals the value of a specified fault location, a random bit is flipped in the value produced by the current instruction.

Once a fault is injected, the execution sequence is monitored to see if a detector location is reached. If so, the value at the detector location is written to a file for offline comparison with the derived detectors for the application. This process is continued until the application ends. Note that only a single fault is injected in each execution of the application. This is because a transient fault is likely to occur at most once during an execution.

Since the simulator does not model the operating system or other aspects of a real system, such as virtual memory management, we modified the simulator to more accurately represent real-world counterparts. This is done by translating the errors detected by the simulator to their corresponding real-world consequence using the mapping in Table 6. The simulator has been calibrated by injecting faults in the real system and comparing it to the simulated system [14].

Table 6: Types of errors detected by simulator and their real-world consequences

Type of Error	Consequence	Simulator Detection Mechanism
Invalid Memory Access	Crash (SIGSEGV)	Consistency checks on address range
Memory alignment Error	Crash (SIGBUS)	Check on memory address alignment
Divide-by-Zero	Crash (SIGFPE)	Check before divide operation
Integer Overflow	Crash (SIGFPE)	Check after every integer operation
Illegal Instruction	Crash (SIGILL)	Check instruction validity before decoding
System Call Error	Crash (SIGSYS)	None, as simulator executes system calls on behalf of application
Infinite Loops	Program Hang (live-lock)	Program executes of double the number of instructions in the golden run
Indefinite Wait	Program Hang (deadlock)	Program execution takes five times more time than the golden run
Incorrect Output	Fail-Silent Violation	Compare outputs at the end of the run

5.3 Experimental Procedure

The experiment is divided into four parts:

1. **Placement of detectors and instrumentation of code.** The dynamic instruction trace of the program is obtained from the simulator and the Dynamic Dependence Graph (DDG) is constructed from the trace. The detector placement points (both variables and locations) are chosen based on the technique described in [14]. For each application, up to 100 detector points are chosen by the analysis, which corresponds to less than 5% of static instructions in the assembly code of the benchmark programs (excluding libraries).
2. **Deriving the detectors based on training set.** The simulator records the values of the selected variables at the detector locations for representative inputs. The dynamic values obtained are used to derive the detectors based on the algorithm in Section 3.1. The training set consists of 200 inputs, which are randomly sampled from a test suite consisting of 1000 inputs for each program. These test suites are provided as part of the Siemens benchmark suite [21].
3. **Fault-injections and coverage estimation.** Fault-injection experiments are performed by flipping single bits in data-values chosen at random from the set of all data values produced during the course of the program's execution. After injecting the fault, the data values at the detector locations are recorded and the outcome of the simulated program is classified as a crash, hang, fail-silent violation, or success (benign). The values recorded at the detector locations are then checked offline by the derived detectors to assess their coverage. The coverage of a detector is expressed in terms of the type of program outcome it detects, e.g., a detector is said to detect a program crash if the program would have crashed had the detector not detected the error. In case the detector does not detect the error at all, its coverage is counted as zero for all four outcome categories.
4. **Computation of false positives.** The application code instrumented with the derived detectors is executed for all 1000 inputs, including the 200 that were used for training. No faults are injected in these runs. If a detector detects an error, then that input is considered a false positive, as there was no injected error but an alarm was raised. We assume that there are no residual errors in the test suite used for training.

For the fault-injection experiments, each application is executed with over 10 inputs chosen at random from those used in the training phase. For each input, 1000 locations are chosen at random from the data values produced by the application. A fault-injection run consists of a single bit-flip (chosen at random) in the one of the 1000 locations. We perform 5 runs for each application-input combination, which corresponds to a total of 50,000 fault-injection runs per application.

6 Results

This section presents the results of the evaluation performed in Section 5.

6.1 Detection Coverage of Derived Detectors

The coverage of the detectors derived using the algorithm in Section 3.1 is evaluated using fault-injections. Figure 5, Figure 6, and Figure 7 show the coverage for crashes, fail-silence violations (FSVs) and hangs obtained for the target applications (expressed as percentages) as a function of the number of detectors placed in each application, ranging from 1 to 100. The following trends may be observed from Figure 5, Figure 6, and Figure 7. The coverage for each type of failure increases as the number of detectors increases, but less than linearly, as there is an overlap among the errors detected by the detectors. Further, the individual error coverage of the derived detectors depends on the type of failure (crash, FSV, hang) detected and the application. In general, crashes exhibit the highest coverage, followed by FSVs and hangs.

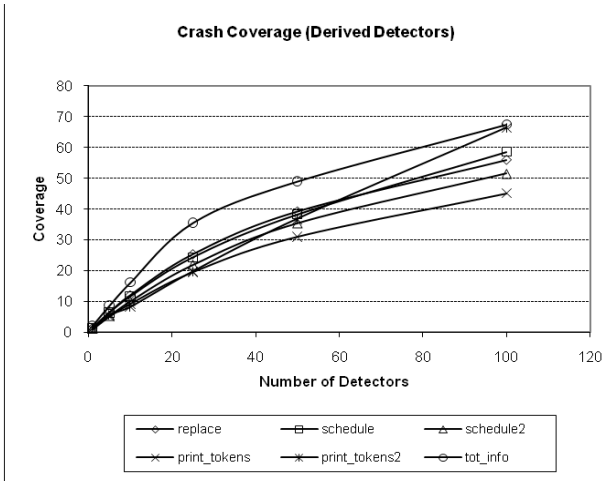

Figure 5: Crash coverage of derived detectors

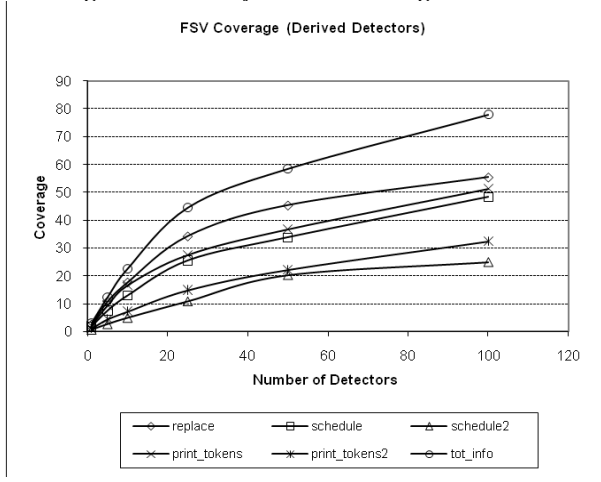

Figure 6: FSV coverage of derived detectors

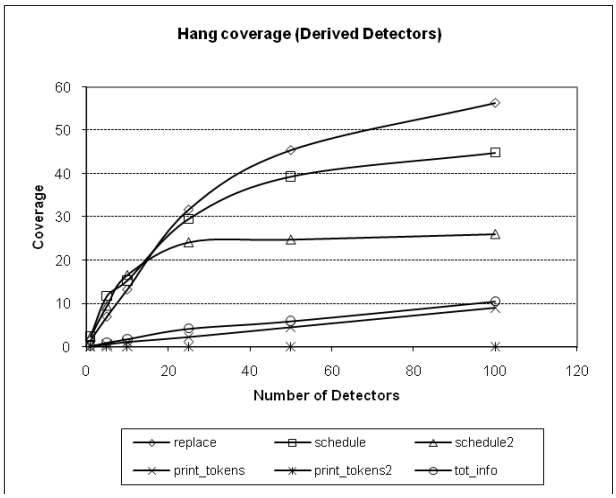

Figure 7: Hang coverage of derived detectors

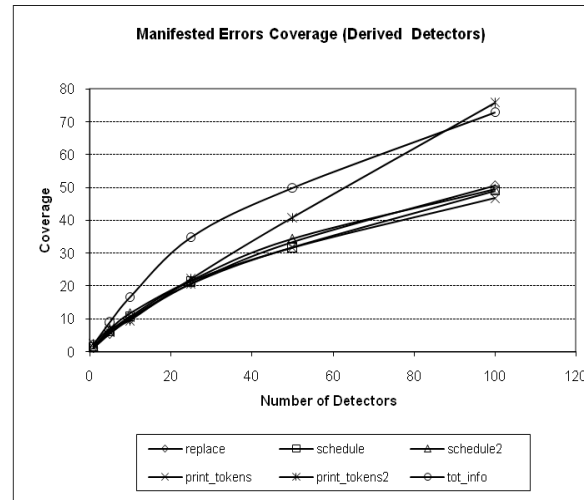

Figure 8: Total coverage of derived detectors

Table 7: Range of detection coverage for 100 detectors

Type of Failure	Minimum Coverage	Maximum Coverage
Program Crash	45% (*print_tokens)*	65% (*tot_info*)
Fail-Silent Violation (FSV)	25% (*schedule2*)	75% (*tot_info*)
Program Hang	0% (*print_tokens2*)	55% (*replace*)
Program Failures	50% (replace, schedule2, print_tokens, tot_info)	75% (schedule, print_tokens2)

Figure 8 shows the percentage of total manifested errors (crash, hang, and FSV) detected by the derived detectors. This is obtained by weighing the detection coverage for the individual failure categories (in Figure 5, Figure 6, and Figure 7) with the fraction of observed errors that result in the failure category (not shown in the figures). The coverage obtained for each type of failure when 100 detectors are placed in each the application is summarized in Table 7. The derived detectors can detect 50- 75% of the errors that manifest in the application. This is because the majority of errors that manifest in an application result in crashes (70-75%) and the rest in fail-silent violations (20-30%) and hangs (0-5%). Hence the coverage for the total manifested errors is dominated by crashes.

The results for coverage correspond to errors that occur in any data value used within the program, not just for errors that occur in the data values checked by the detector. For example, if even a single bit-flip occurs in a single instance of any data value used in the program and this error results in a program crash, hang, or fail-silence violation, then one of the 100 detectors placed in the program will detect the error 50-75% of the time. As mentioned in Section 5.1, 100 detectors correspond to less than 5% of program locations in the program's assembly code, not including library functions.

6.2 False Positives

False positives can occur when a detector flags an error even if there is no error in the application. A false positive for an input can occur when the values at the detector points for the input do not obey either the detector's rule or the exception condition learned from the training inputs. This occurs if the training set is not comprehensive enough, i.e., it does not cover all the values that may be exhibited by a variable checked by a detector.

The training set for learning the detectors consists of 200 inputs, and the false positives are computed across 1000 inputs for each application. No faults were injected in these runs. Therefore, any alarm raised by the detectors for any of the 1000 inputs is a false positive. *If even a single detector detects an error for a particular input, then the entire input is treated as a false positive, even if no other detector detects an error for the input.*

Figure 9 presents the percentage of false positives for each of the target applications as a function of the number of detectors placed in the program. Across all applications, the false positives are no more than 2.5% when 100 detectors are placed in the program. For the *replace, schedule2, print_tokens,* and *print_tokens2* applications, the false positives are observed in less than 1% of the inputs, while for the *schedule* and *tot_info* applications, the false positives are observed in about 2% of the inputs. While the number of false positives increases as the number of detectors increases, it reaches a plateau as the number of detectors is increased beyond 50. This is because a false positive input is likely to trigger multiple detectors once the number of detectors passes a certain critical threshold (which occurs at around 50 detectors in the benchmark programs). However, no such plateau was reached for the coverage results in Figure 8, even up to 100 detectors.

Effect of False Positives. When a detector raises an alarm, we need to determine whether an error was really present or whether it is a false positive. If the error was caused by a transient fault (as we assume in this paper), then it is likely to be wiped out when the program is re-executed. If, on the other hand, the detection was a false positive and hence a characteristic of the input given to the program, the detector will raise an alarm again during re-execution. In this case, the alarm can be ignored, and the program be allowed to continue. Thus, the impact of a false positive is essentially a loss in performance due to re-execution overhead. Since the percentage of false positives is less than 2.5%, the re-execution overhead is small.

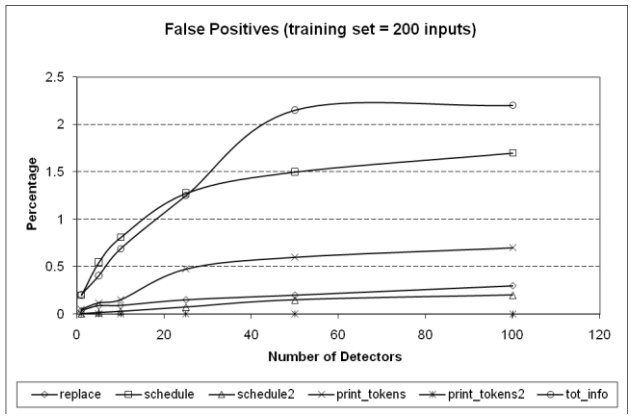

Figure 9: Percentage of false positives for 1000 inputs

Note that rollback recovery or re-execution may not always be possible in certain systems. For example, in real-time systems, re-execution can lead to missed deadlines, and in distributed systems, it may trigger system-wide rollback. In such systems, false positives may cause an impact that goes beyond loss of performance, for example, violations of specifications. Recovery techniques are outside the scope of this study, hence we assume that rollback recovery is both feasible and results in only a (modest) performance overhead. Similar assumptions have been made by prior work [23].

6.3 Effect of Training Set Size

The results reported in Section 6.1 and 6.2 for coverage and false positives of the derived detectors used a training set of 200 inputs from a total of 1000 inputs for each application. In this section, we consider the effects of varying the size of the training set from 100 inputs to 300 inputs. In these experiments, the number of detectors in each application is fixed at 100, and the coverages for different kinds of failures and false positives are evaluated for each application. The results are shown in Figure 10, Figure 11, Figure 12, and Figure 13.

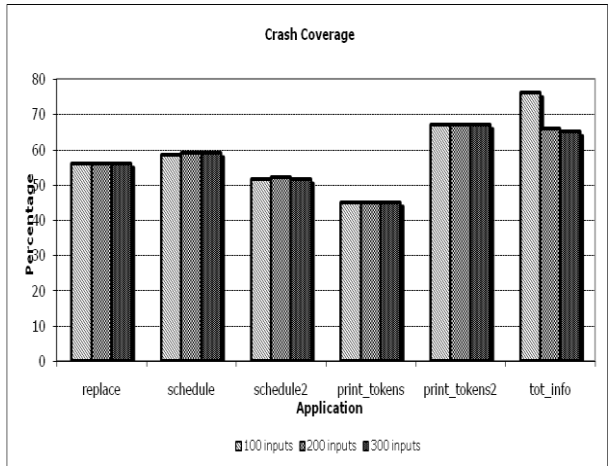

Figure 10: Crash coverage for different training set sizes

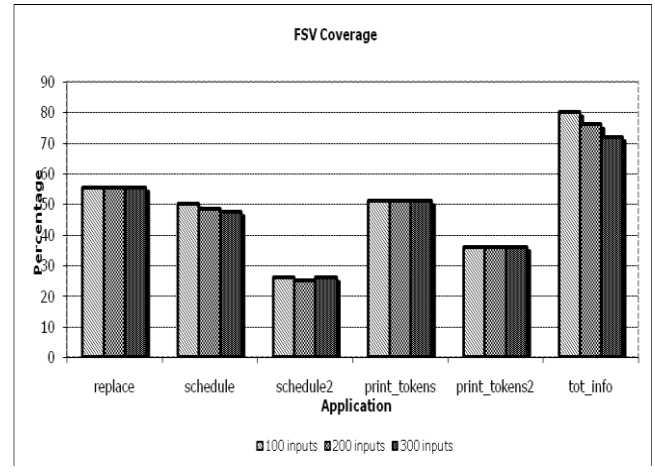

Figure 11: FSV coverage for different training set sizes

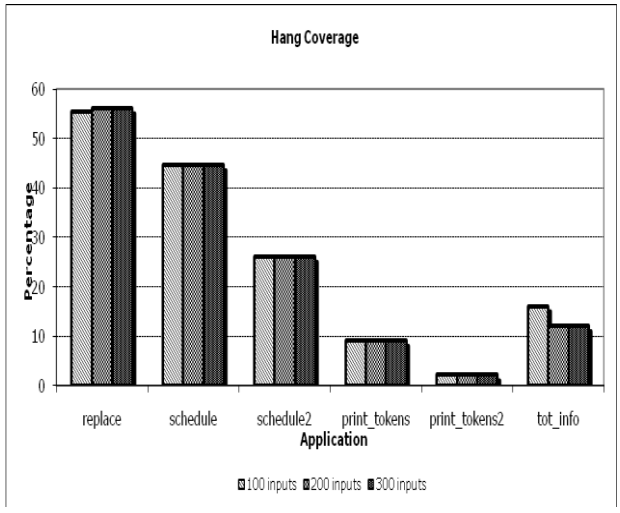

Figure 12: Hang coverage for different training set sizes

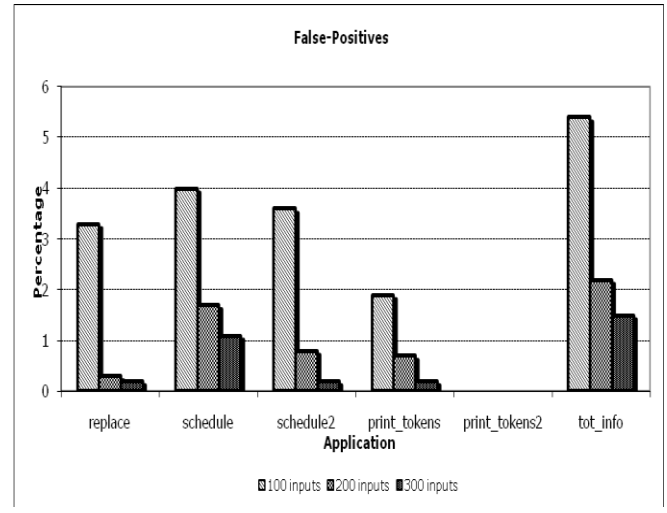

Figure 13: False positives for different training set sizes

The results from the graphs are:

- The false positives decrease from 5% to 2% as the training set size is increased from 100 inputs to 200 inputs, and to less than 1% for 300 inputs across all programs except *tot_info* (for which the false positives are 1.5% for 300 inputs).
- The coverages for crashes and hangs remain constant as the training set size increases for all applications (Figure 8, Figure 10), except in the case of *tot_info,* where the coverage first decreases from 100 to 200 inputs and then remains constant from 200 to 300 inputs (for crashes and hangs).
- The coverage for fail-silent violations decreases marginally as the size of the training set increases from 100 inputs to 300 inputs (Figure 9). This decrease in fail-silent violations is less than 2% for all applications except *tot_info* (5%).

Therefore, increasing the training set size from 100 to 200 inputs decreases the false positives significantly, while increasing it from 200 to 300 inputs does not have as large an impact on false positives. However, the impact on the detection coverage from increasing the training set size is minimal. This suggests that the detectors, once learned, are relatively stable across different inputs, and that their detection capabilities are not affected by the input beyond a certain number of training inputs (200).

Note that different training set sizes may also influence the code coverage achieved. In our experiments, the test suites had sufficiently high code coverage that this was not a major issue. However, it is possible in other applications that some test suites have lower code coverage than others, which may reduce their error detection coverage. In such cases, care must be taken to ensure that training sets have approximately similar code coverage.

Another consequence of choosing different training set sizes is that it introduces differences in the execution time of the detectors. We found such differences to be marginal in terms of the overall execution time of the program. Nonetheless, this is a potential issue in some applications.

6.4 Comparison with Best-Value Detectors

As seen in Section 6.1, the derived detectors detect about 45-65% of crashes, 25-80% of fail-silent violations, and 0-55% of hangs in a program. This section investigates why the remaining errors are not detected and how the detectors can be improved. To form the basis of the discussion, we consider a hypothetical detector that keeps track of the entire history of data values observed at a detector location and uses this knowledge to flag an error. We call this a *best-value detector*, as it represents the maximum coverage that can be obtained by a single value-based detector (including one written by a programmer). *This section represents the main contribution of this paper over and above our previous work [13].*

The best-value detector may not be achievable in practice, as in addition to requiring enormous space and time overheads (to store the entire history of values), it assumes apriori knowledge of all possible inputs to the program. Nonetheless, the coverage of the best-value detector provides an upper bound on the coverage that can be obtained with data-value based detectors, such as the detectors considered in this paper. Further, it provides insights into improving the coverage of the derived detectors, which is the main motivation for this study.

We build the best-value detector by executing the program under a specific set of inputs and storing the entire sequence of values observed at each location at which a detector is placed. This fault-free execution is referred to as the *golden run* of the program. Faults are injected into the program, and the values of the detector locations are recorded. An offline post-processing phase compares each value at the detector location with the value recorded in the golden run. If the program completed execution under the fault, the entire value sequence is compared and any deviation is reported as a successful detection. If the program crashes (hangs), only mismatches in the values that were recorded before the crash (hang) are reported as detections.

In this study, the number of detectors in the program is fixed at 100, which is the maximum number of detectors considered in the previous studies (Section 6.1 and 0). For each application, both the best-value detectors and the derived detectors are placed at the same variables and locations. For fault-injection, the program is executed under the same set of inputs (10 in this study) that were used to derive the best-value detectors. The same set of faults is injected for evaluating both the best-value detectors and the derived detectors.

Figures 14 through 17 show the coverage obtained with the best value detectors for crashes, fail-silent violations, and hangs. The corresponding coverage obtained by the derived detectors (for 100 detectors) is also shown in the graphs for ease of comparison.

The results of the comparison are as follows:

Crashes. Compared to the best-value detectors, the derived detectors detect between 75% (*replace*) and 100% (*schedule2, print_tokens2*) of errors that result in crashes (Figure 14)

FSVs. Compared to the best-value detectors, the derived detectors detect between 40% (*print_tokens2*) and 85% (*tot_info*) errors that result in fail-silent violations (Figure 15).

Hangs. Compared to the best-value detectors, the derived detectors detect between 50% (*tot_info)* and 100% (*schedule2, print_tokens2)* for program hangs (Figure 16).

Note that the best-value detectors do not obtain 100% coverage for any of the failure categories. This is because they are placed only at critical variables in the application, and not all errors in the application may propagate to the critical variable. Further, the best-value detectors do not include timing information, hence they may not detect changes in the control-flow of the application unless there is a corresponding change in the application's data values. This is the main difference between the best-value detectors and the *ideal detectors* introduced in our earlier study [14], which keep track of both data and timing information to detect errors. The ideal detectors achieved coverage of between 90% and 100% when 100 detectors were placed in the program. We do not consider ideal detectors in this paper.

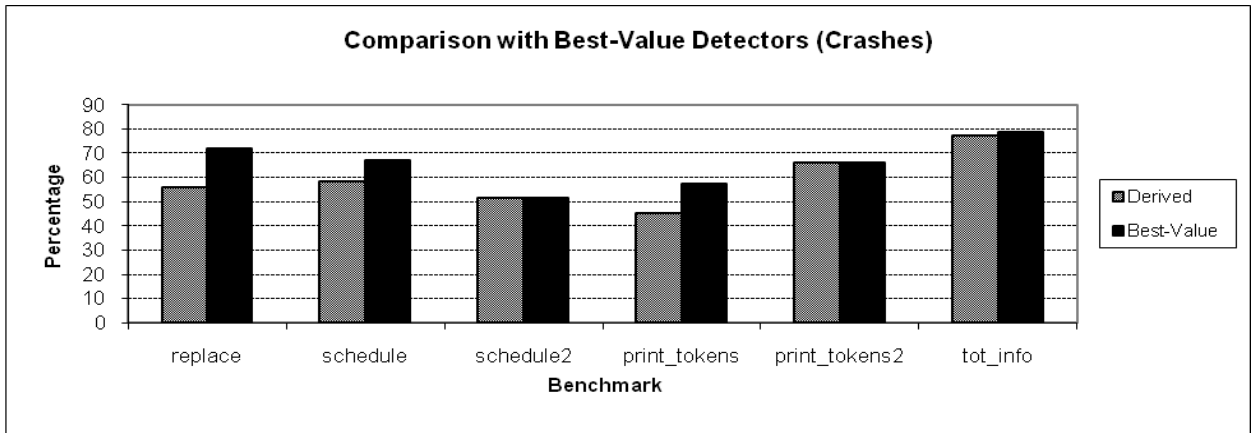

Figure 14: Comparison between best-value detectors and derived detectors for crashes

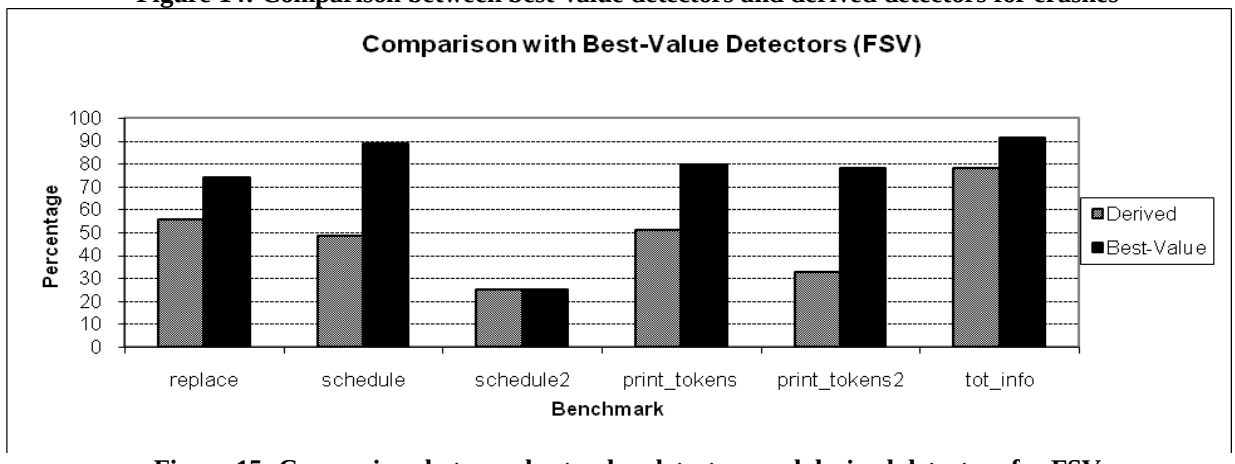

Figure 15: Comparison between best-value detectors and derived detectors for FSV

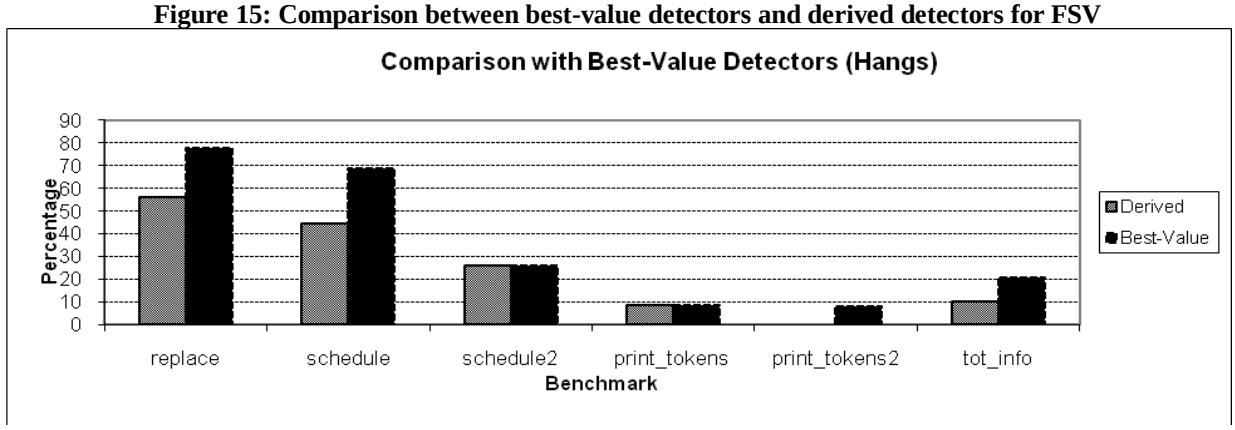

Figure 16: Comparison between best-value detectors and derived detectors for hangs

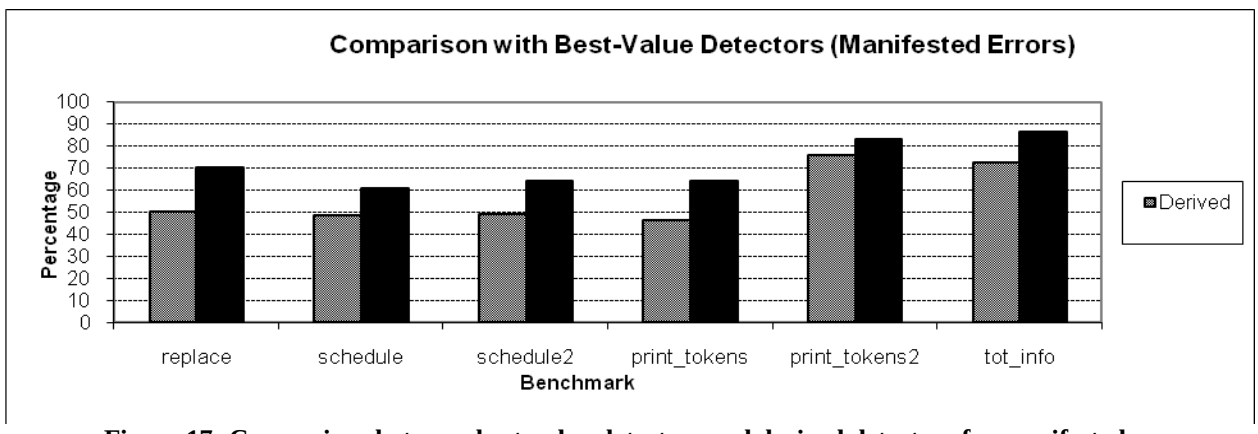

Figure 17: Comparison between best-value detectors and derived detectors for manifested errors

Figure 17 shows the coverage obtained by the derived detectors vis-à-vis the best-value detectors for all manifested errors in the application. The derived detectors achieve between 70% (*replace)* and 90% (*print_tokens2)* of the coverage provided by the best-value detectors. The overall coverage is high because the total number of manifested errors is dominated by crashes, for which the derived detectors obtain high coverage relative to the best-value detectors. This is because crashes are often caused by egregious corruptions of data values, which are easily detected by the derived detectors.

However, derived detector coverage for Fail-Silent Violations (FSVs) is not as high as the corresponding coverage of the best-value detectors. This is because subtle violations in data values may not be detected easily by the derived detectors. The best-value detectors are tailored for each input (based on the golden run of the application for the input) and hence have 100% knowledge of the application's data values for that input. In contrast, the derived detectors should be general across inputs, otherwise they will incur false positives. This suggests that FSVs are caused by corruptions in data values that are invalid for the given input but may be valid for a different input. Since the derived detectors are not input-specific, they have no way of distinguishing an incorrect value from the correct value. Therefore one way of improving the coverage of derived detectors with respect to FSVs is to make them input-specific. This is an avenue for future work.

Finally, the coverage of the best-value detectors for hangs exhibits wide variation across applications. As mentioned earlier, hangs are often caused by changes in the control-flow of the application, which is not detected by either the best-value detector or the derived detector (unless it causes a data error). As a result, the best-value detectors exhibit high levels of hang coverage only for those applications in which the control-flow is highly dependent on its data values (e.g., replace and schedule). This can be remedied by including control-flow or timing information in the detectors, a direction for future work.

6.5 Hardware Implementation Results

The proposed design of the DLX processor, the RSE Interface, and the Error Detector Modules for different applications were synthesized using Xilinx ISE 7.1 tools targeting a Xilinx Virtex-E FPGA. The Xilinx Virtex series of FPGAs consists mainly of several types of logic cells: (1) 4-input Look-Up Tables (LUTs) statically programmed during the bootstrap with the configuration bit-stream, (2) flip-flops (FFs), storage elements in the user visible system state, and (3) Block RAM (BRAMs), which are memory blocks that can store up to 4096 bits. Four LUTs and four FFs compose a logic unit called *Slice*.

Area and Clock Period Overhead. Table 8 reports the synthesis results in terms of area (i.e., FFs, LUTs, BRAMs, and total Slices) and minimum clock frequency, for the unmodified DLX processor and the complete RSE Interface. It can be observed that the RSE interface has an area overhead of 14.9% over the unmodified DLX processor and a negligible impact on its clock period.

Table 9 shows the synthesis results in terms of area and minimum clock period for each of the benchmarks considered in Section 5.1. The benchmark name is shown in the first column, and the number of (unique) detectors synthesized for the program is shown in the second column. The third, fourth, and fifth columns report the number of flip-flops, the number of BRAMs, and the number of LUTs respectively. Column 6 reports the number of slices, while column 7 reports the maximum clock period achieved during synthesis. Finally, columns 8 and 9 report the overheads of the EDM in terms of the percentage of extra slices, with and without the RSE.

Table 8: Area and timing results for the DLX processor and the RSE Framework

	FFs	LUTs	BRAMs	Slices	Clock Period (ns)
DLX processor	4873	16395	0	9526	58.8
RSE Interface	2465	2329	0	1420	2.01

The results for the area overheads in Table 9 are that the number of slices required for the implementation of the EDM ranges is between 2685 and 2911 and that the number of additional BRAMs required is 9.

Performance Overhead. The performance overhead incurred due to the extra hardware is calculated as:

$$Overhead = [\ (Total\ clock\ cycles + Extra\ clock\ cycles) * (T_{with\ EDM} - T_{without\ EDM})\ ] / (Total\ Clock\ Cycles * T_{without\ EDM})$$

where $T_{with\ EDM}$ and $T_{without\ EDM}$ are the clock cycle times with and without the EDM respectively, and *Extra clock cycles* is the number of additional clock cycles to execute the code instrumented with the CHECK instructions.

Due to space constraints, we only report results for the benchmark with the highest overheads, namely *schedule2*. For this program, The number of extra clock cycles is 594, while the total number of clock cycles is nearly 1 million (the exact value does not matter in the above calculation), $T_{with\ EDM}$ is 58.82 ns and $T_{without\ EDM}$ is 55.55 ns. Using the above formula, the total performance overhead is 5.6%.

We obtain such a low overhead because (1) the detectors are executed by the EDM concurrently with the application within a few cycles, hence the latency of the detectors' execution may be overlapped with the execution of the application, and (2) the clock period of the superscalar DLX processor is only marginally impacted by the RSE interface and the EDM.

7 Related Work

Broadly, error detection techniques can be classified based on two criteria: (1) how detectors are derived (static or dynamic) and (2) how checking is performed (static or dynamic). These lead to 4 categories of detectors that span the spectrum of purely static techniques [1-2, 4, 24] to purely dynamic techniques [25-26]. This categorization also includes hybrid techniques in which the detectors are derived statically and checked dynamically [10-11, 27] and those in which the detectors are derived dynamically but checked statically (for example, DAIKON [3]). These techniques are described in further detail in Table 10.

Table 9: Area and timing overheads across benchmarks

Benchmark Name	Number of Unique Detectors	FFs	BRAMs	LUTs	Slices	Clock Period [ns]	EDM Slice Overhead [%]	EDM + RSE Interface Slice Overhead [%]
tot_info	91	2913	9	5174	2685	20.7	28.2	43.1
replace2	91	2913	9	5176	2686	21.6	28.2	43.1
print_tokens	98	3169	9	5575	2876	19.7	30.2	45.1
print_tokens2	98	3169	9	5578	2875	21.1	30.2	45.1
schedule	98	3169	9	5578	2875	20.4	30.2	45.1
schedule2	99	3201	9	5626	2911	19.9	30.6	45.5

Table 10: Related techniques

Technique	Description	Comments
Prefix [1]	Uses symbolic execution through selected paths in a program to find known kinds of errors (e.g., NULL pointer dereferences).	1. Requires programmer to write annotations in the source code. 2. Has high false positive rate due to infeasible paths.
C-Cured [24]	Verifies that pointers do not write outside their intended memory objects, thereby ensuring memory safety.	1. Protects only against errors that violate memory safety – does not protect computation errors. 2. Does not handle hardware errors or errors originating in unverified code.
LCLINT [2]	Checks whether a program conforms to its specification and whether it adheres to predefined programming rules.	1. Requires programmer to provide specifications or write annotations in code. 2. Finds only errors that violate the predefined rules.
Engler et al. [4]	Analyzes source files to find application-specific programming patterns and identifies violations of the patterns as bugs.	1. May incur false positives i.e., the violation of the pattern may not necessarily be a bug. 2. Does not handle runtime errors or hardware faults.
DAIKON [3]	Infers invariants from the dynamic execution of program based on representative inputs.	1. Does not take placement of detectors into account - program may crash before the execution reaches the detector location. 2. Requires programmer to interpret the invariants and locate bugs and filter out false-invariants.
Voas et al. [27]	Considers a general methodology to embed detectors in programs to detect errors. Characterizes properties of good detectors.	1. Does not consider how to derive the detectors. 2. Detector placement methodology relies heavily on programmer's knowledge of application.
Rela et al. [10]	Evaluates the coverage provided by existing assertions in a program vis-à-vis control-flow error detection techniques or algorithm-based, fault-tolerance techniques.	Does not consider deriving or embedding assertions in a program. Assumes that assertions have already been inserted by programmer during program development (for debugging).
Hiller et al. [11]	Places error detectors in an embedded system to detect data errors. Considers different classes of detectors based on properties of the signals in an embedded system, and the detectors are placed in the system to maximize the coverage.	1. Programmer needs to specify class and parameters of each detector - detector derivation is not automated. 2. Detector placement is based on extensive fault-injections, which are time-consuming and effort-intensive.
DIDUCE [25]	Uses software anomaly detection to locate corner cases and find bugs. Formulates strict hypothesis about program behavior in the beginning of the execution and gradually relaxes the assumptions as program executes to learn new behavior.	1. Program may crash before reaching detector point, and the error will not be detected or may skip detection. 2. Does not address errors that occur when the invariants are being learned (at the beginning of program execution).
Maxion et al. [26]	Characterizes the generic space of anomaly detectors for embedded applications.	Does not define specific types of error detectors or explain how they are derived from the application.

The work closest to ours is Hiller et al. [11], which manually derives detectors for an embedded system using rule-based templates. They obtain detection coverage of about 80% with 7 assertions for (random) errors that cause failure in their embedded system application. However, in their study, about 2000 errors are injected into the system during a short period of 40 seconds, and if one of their executable assertions detects one of the errors in this period, it is considered a successful detection. In contrast, *we inject only a single error* in each run. Further, 7 out of 24 signals are targeted for detection in the embedded system considered in [11], which corresponds to about 30% of the system. In comparison, we place detectors in only 5% of the application code.

In earlier work [13], we outlined a methodology to derive error detectors automatically based on dynamic execution traces of a program. The main difference between that paper and this one is Section 6.4, which compares the coverage provided by the derived error detectors with that provided by the best-value detectors. The comparison provides valuable insights into what errors are missed by the detectors and how to improve detector coverage.

Since we published the work in [13], three papers have been published based on the idea of using dynamically derived program invariants for runtime error detection [28-30]. These papers use online or offline profiling of the program to derive value-based invariants and use special hardware to check them at runtime. The Appendix discusses the differences between these papers and ours.

8 CONCLUSIONS

This paper proposed a novel technique for preventing a wide range of data errors from corrupting the execution of an application. This technique consists of an automated methodology to derive fine-grained, application-specific error detectors using an algorithm based on dynamic traces of application execution. A set of error detector classes, parameters, and locations, were derived in order to maximize the error detection coverage for a target application. The paper also presented an automatic framework for synthesizing the detectors in hardware to ensure low-latency, concurrent error detection. The coverage of the derived detectors was evaluated using fault-injections and found to be about 50-75% for failure-causing errors. The area and performance overheads of the detectors were about 15% and 5%, respectively.

Acknowledgments. This work was supported in part by National Science Foundation (NSF) grants CNS-0406351 and CNS-0524695, the Gigascale Systems Research Center (GSRC/MARCO), Motorola Corporation as part of the Motorola Center for Communications (UIUC), and Intel Corporation. We thank Fran Baker for editorial support.

REFERENCES

[1] W. R. Bush, *et al.*, "A static analyzer for finding dynamic programming errors," *Software Practice and Experience*, vol. 30, pp. 775-802, 2000.

[2] D. Evans, *et al.*, "LCLint: a tool for using specifications to check code," in *2nd ACM SIGSOFT symposium on Foundations of software engineering*, New Orleans, Louisiana, United States, 1994, pp. 87-96.

[3] M. D. Ernst, *et al.*, "Dynamically discovering likely program invariants to support program evolution," in *21st international conference on Software engineering*, Los Angeles, California, United States, 1999, pp. 213-224.

[4] D. Engler, *et al.*, "Bugs as deviant behavior: a general approach to inferring errors in systems code," in *Eighteenth ACM Symposium on Operating systems principles*, Banff, Alberta, Canada, 2001, pp. 57-72.

[5] W. Gu, *et al.*, "Characterization of linux kernel behavior under errors," in *International Conference on Dependable Systems and Networks*, 2003, pp. 459-468.

[6] C. Basile, *et al.*, "Group communication protocols under errors," in *22nd International Symposium on Reliable Distributed Systems*, 2003, pp. 35-44.

[7] I. Lee and R. K. Iyer, "Software dependability in the Tandem GUARDIAN system," *IEEE Transactions on Software Engineering*, vol. 21, pp. 455-467, 1995.

[8] D. Andrews, "Using executable assertions for testing and fault tolerance,," in *9th Faul-tolerance Computing Symposium*, Madison, WI, 1979, p. 21.

[9] N. G. Leveson, *et al.*, "The use of self checks and voting in software error detection: an empirical study," *IEEE Transactions on Software Engineering*, vol. 16, pp. 432-443, 1990.

[10] M. Z. Rela, *et al.*, "Experimental evaluation of the fail-silent behaviour in programs with consistency checks," in *Annual Symposium on Fault-tolerant Computing*, Sendai, 1996, pp. 394-403.

[11] M. Hiller, "Executable Assertions for Detecting Data Errors in Embedded Control Systems," in *International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, 2000, p. 24.

[12] M. Hiller, *et al.*, "On the Placement of Software Mechanisms for Detection of Data Errors," in *International Conference on Dependable Systems and Networks*, 2002, pp. 135-144.

[13] K. Pattabiraman, *et al.*, "Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware," in *Sixth European Dependable Computing Conference*, Coimbra, Portugal, 2006, pp. 97-108.

[14] K. Pattabiraman, *et al.*, "Application-based metrics for strategic placement of detectors," in *Pacific Rim Dependable Computing*, Changsha, China, 2005, pp. 95-102.

[15] J. Ohlsson, *et al.*, "A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog," in *Twenty-Second International Symposium on Fault-tolerant Computing*, 1992, pp. 316-325.

[16] N. Mehdizadeh, *et al.*, "Analyzing fault effects in the 32-bit OpenRISC 1200 microprocessor," in *Third International Conference on Availability, Reliability and Security (ARES)*, 2008, pp. 648-652.

[17] J. Gray, "Why do computers stop and what can be done about it," in *Symposium on Reliable Distributed Systems*, 1986, pp. 3–12.

[18] J. Voas, "Software testability measurement for intelligent assertion placement," *Software Quality Control*, vol. 6, pp. 327-336, 1997.

[19] N. Nakka, *et al.*, "An Architectural Framework for Providing Reliability and Security Support," in *International Conference on Dependable Systems and Networks*, 2004, p. 585.

[20] D. A. Patterson and J. L. Hennessy, *Computer architecture: a quantitative approach*: Morgan Kaufmann Publishers Inc., 1990.

[21] M. Hutchins, *et al.*, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *16th international conference on Software engineering*, Sorrento, Italy, 1994, pp. 191-200.

[22] T. Austin, *et al.*, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, vol. 35, pp. 59-67, 2002.

[23] N. J. Wang and S. J. Patel, "ReStore: Symptom-Based Soft Error Detection in Microprocessors," *IEEE Trans. Dependable Secur. Comput.*, vol. 3, pp. 188-201, 2006.

[24] G. C. Necula, *et al.*, "CCured: type-safe retrofitting of legacy code," in *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Portland, Oregon, 2002, pp. 128-139.

[25] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *24th International Conference on Software Engineering*, Orlando, Florida, 2002, pp. 291-301.

[26] R. A. Maxion and K. M. C. Tan, "Anomaly Detection in Embedded Systems," *IEEE Trans. Comput.*, vol. 51, pp. 108-120, 2002.

[27] M. V. Jeffrey and W. M. Keith, "The Avalanche Paradigm: An Experimental Software Programming Technique for Improving Fault-tolerance," presented at the Proceedings of the IEEE Symposium and Workshop on Engineering of Computer Based Systems, 1996.

[28] P. Racunas, *et al.*, "Perturbation-based Fault Screening," presented at the Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, 2007.

[29] M. Dimitrov and H. Zhou, "Unified Architectural Support for Soft-Error Protection or Software Bug Detection," presented at the Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, 2007.

[30] S. Sahoo, *et al.*, "Using likely program invariants to detect hardware errors," in *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, Anchorage, AK, 2008, pp. 70-79.

Authors' Biographies

Karthik Pattabiraman received the M.S and PhD degree in computer science from the University of Illinois at Urbana-Champaign (UIUC) in 2004 and 2009. He is currently an assistant professor at the University of British Columbia in electrical and computer engineering. His research interests include design of reliable and secure applications using static and dynamic analysis, as well as experimental and formal techniques for dependability validation. Karthik's dissertation proposed the idea of application-aware dependability and he was the lead graduate student in the Trusted Illiac project at the University of Illinois. Based on his dissertation work, Karthik Pattabiraman was awarded the *William C. Carter* award in 2008 by the IFIP Working Group on Dependability (WG 10.4) and the IEEE Technical Committee on Fault-tolerant Computting (TC-FTC). He is a member of the IEEE and IEEE Computer Society.

Dr. Giacinto Paolo Saggese received the M.S. (Summa cum Laude) in Electrical Engineering from the University of Naples in 2000 and the Ph.D. in Electrical and Computer Engineering from University of Naples in 2004. From 2003 to 2004 he was at University of Illinois at Urbana Champaign (UIUC) as a visiting scholar completing his PhD thesis on custom hardware for cryptographic and secure applications. From 2004 to 2005 he was a Post Doc at UIUC doing research on soft errors and automatic application of reliability-enhancing techniques. From 2006 to 2007 he was a Verification Architect at NVIDIA Corp. In 2007 he co-founded ZeroSoft Inc., a start-up developing innovative tools for verification, serving as VP of Engineering. In 2010, after ZeroSoft technology was successfully adopted by major semiconductor companies, ZeroSoft was acquired by Synopsys Inc, where Giacinto Paolo is now a Principal Engineer.

Daniel Chen received his BS degree in computer engineering from University of Illinois at Urbana-Champaign in 2005. He subsequently received the MS degree in computer engineering from University of Illinois at Urbana-Champaign in 2008. He is currently working as a visiting computer engineer for the Center for Reliable and High-Performance Computing at the University of Illinois at Urbana-Champaign. His research interests include computer security and reliability.

Dr. Zbigniew T. Kalbarczyk is currently Research Professor at the Center for Reliable and High-Performance Computing in the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. Dr. Kalbarczyk's research interests are in the

area of design and validation of reliable and secure computing systems. Currently, he is a lead researcher on the project to explore and develop high availability and security infrastructure (including use of dedicated software and reprogrammable hardware) capable of managing redundant resources to foil security threats, detect errors in both the user applications and the infrastructure components, and recover quickly from failures when they occur. His research involves also designing of techniques for automated validation and benchmarking of dependable computing systems using formal (e.g., model checking) and experimental methods (e.g., fault/attack injection). He served as a program Chair of Dependable Computing and Communication Symposium (DCCS), a track of the International Conference on Dependable Systems and Networks (DSN) 2007 and Program Co-Chair of Computer Performance and Dependability Symposium, a track of the DSN 2002. Dr. Kalbarczyk has published over 90 technical papers and is regularly invited to give tutorials and lectures on issues related to design and assessment of complex computing systems. He holds PhD degree in computer science from the Technical University of Sofia, Bulgaria. He is a member of the IEEE, the IEEE Computer Society, and IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance.

Ravishankar K. Iyer is Interim Vice Chancellor for Research at the University of Illinois at Urbana-Champaign, where he is a George and Ann Fisher Distinguished Professor of Engineering. He holds appointments in the Department of Electrical and Computer Engineering and the Department of Computer Science. He is Director of the Center for Reliable and High-Performance Computing at the Coordinated Science Laboratory and Chief Scientist at the Information Trust Institute. Iyer's research interests are in the area of dependable and secure systems. He has been responsible for major advances in the design and validation of dependable computing systems. He currently leads the TRUSTED ILLIAC project at Illinois, which is developing application-aware adaptive architectures for supporting a wide range of dependability and security requirements in heterogeneous environments. Professor Iyer is a Fellow the AAAS, the IEEE and the ACM. He has received several awards including the Humboldt Foundation Senior Distinguished Scientist Award for excellence in research and teaching, the AIAA Information Systems Award and Medal for "fundamental and pioneering contributions towards the design, evaluation, and validation of dependable aerospace computing systems," and the IEEE Emanuel R. Piore Award "for fundamental contributions to measurement, evaluation, and design of reliable computing systems."