Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware

K. Pattabiraman, G.P. Saggese, D. Chen, Z. Kalbarczyk, R.K. Iyer

Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign

{pattabir, saggese, dchen8, kalbar, iyer}@crhc.uiuc.edu

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. An algorithm based on dynamic traces of application execution is developed for extracting the set of error detector classes, parameters, and locations in order to maximize the error detection coverage for a target application. 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 execution. Coverage (evaluated using fault injection) of the error detectors derived using the proposed methodology, the additional hardware resources needed, and performance overhead for several benchmark programs are also reported.*

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 caught by generic techniques such as ECC in memory.

It is common practice for developers to write assertions in programs for debugging and error-detection purposes. For example, Andrews [2] discusses the use of executable assertions (checks for data reasonableness) to support testing and fault-tolerance. Chandra et al. [16] uses analysis of open-source data to show that application-specific knowledge is vital in detecting and recovering from many kinds of application errors.

Many static and dynamic analysis tools (Prefix[5], LCLint [6], Daikon[14]) have been proposed to find bugs in programs. However, these tools are not geared toward detecting runtime errors. 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 chances for an error to propagate is essential because programs, upon encountering an error that could eventually lead to a crash, may execute for billions of cycles before crashing [17]. During this time, the program can exhibit unpredictable behavior, such as writing a corrupted state to a checkpoint [10] or sending a corrupted message to another process [15] (in a distributed environment).

A technique to detect runtime errors was proposed by Hiller et al. [18], who insert assertions in an embedded application based on the high-level behavior of its signals. They facilitate the programmer in inserting assertions by means of well-defined classes of detectors. In a companion paper, they also describe how to place assertions by performing extensive fault-injection experiments [19]. While this technique is successful if the programmer has extensive knowledge of the application and if fault-injection can be performed, it is desirable to enable deriving detectors without such knowledge and inserting them in strategic locations without performing fault-injection.

Our goal is to devise detectors that preemptively capture errors that affect 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 that are automatically derived based on the dynamic behavior of the application.*

In earlier work, we introduced and evaluated a methodology to place detectors in application code to maximize error-detection coverage [9]. That methodology chooses both the program variable and the program location at which the detector must be placed *without* requiring fault-injection into the program. For a large-application such as *gcc*, we showed that by placing detectors at 10 program points according to the methodology described in the paper, it is possible to obtain coverage of up to 80 % (for crash failures). It was assumed that if an error propagated to a variable chosen for placing a detector, then there existed some mechanism that would detect the error in that variable (*ideal detector*).

In this paper, we derive the executable assertions that effectively constitute the specific detectors for the chosen variables (in practice). The method proposed for automated derivation of error detectors is based on analysis of data values produced during the course of a program execution for a set of representative inputs. Error detectors corresponding to each variable and location (chosen according to the detector placement methodology in [9]) are learned based on pre-determined generic classes of rules. The derived detectors continuously monitor the application at run-time, checking for the presence of errors.

The detectors derived can be implemented either in software or hardware. Hardware implementation is essential to ensure low-overhead error detection without sacrificing coverage. An automatic process for converting the detectors from an abstract representation into a hardware implementation is outlined.

The proposed methodology is applied to derive detectors for several benchmark programs. Experimental evaluation of coverage (assessed via random fault injection into application data), the additional hardware resources needed, and performance overheads indicate the following: (1) The derived detectors detect between 50 and 75% of errors that are manifested at the application level when 100 detectors are placed in the application code (corresponding to about 5% of application code). (2) False positives (detectors flag an error when no error is present) are less than 2.5% for the benchmarks considered. (3) Hardware implementation of the detectors results in a performance overhead of less than 5%, with acceptable area and power overheads.

2 Approach and Fault-Model

The derivation and implementation of the error detectors in hardware and software consists of four main phases, depicted in Figure 1. 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 use of the detectors at runtime.

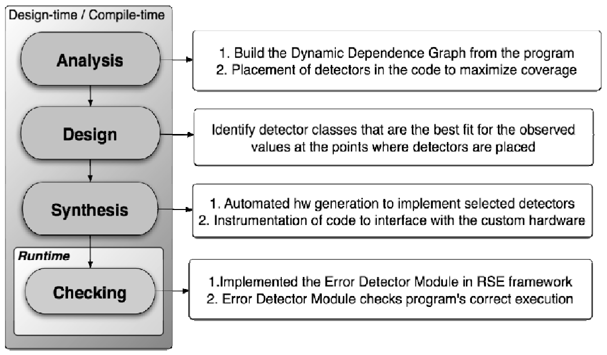
Design-time / Compile-time
Analysis
1. Build the Dynamic Dependence Graph from the program
2. Placement of detectors in the code to maximize coverage
Design
Identify detector classes that are the best fit for the observed values at the points where detectors are placed
Synthesis
1. Automated hw generation to implement selected detectors
2. Instrumentation of code to interface with the custom hardware
Runtime
Checking
1.Implemented the Error Detector Module in RSE framework
2. Error Detector Module checks program's correct execution

Figure 1: Steps in derivation and implementation of error detectors

During the *analysis* phase, the program locations and variables for placing detectors to maximize coverage are identified, based on the execution of the code and the Dynamic Dependence Graph (DDG) of the program. This approach is based on the technique proposed in [9] and does not require fault-injection into the program to choose the detector variables and locations. Rather, it uses metrics such as the fanouts and lifetimes of nodes in the DDG to place the detectors.

The program code is then instrumented to record the values of the chosen variables at the locations selected for detector placement. The recorded values are used during the *design* phase to derive the best detector that matches the observed values for the variable, based on predetermined generic detector classes (see Section 5).

After this stage, the detectors can either be integrated into application code as software assertions or implemented in hardware. The coverage of the derived detectors is quantified in the context of benchmark applications executing on an enhanced version of the Simplescalar simulator [12] (see Section 6). This applies both to software and hardware implementation of the detectors.

The *synthesis* phase converts the generated assertions into an HDL (Hardware Description Language) 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 7. 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 this as an error in the application.

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

The fault-model also represents certain types of software errors that result in data-value corruptions such as: (1) synchronization errors or race conditions that result in corruptions of data values due to incorrect sequencing of operations, (2) memory corruption errors, e.g., buffer-overflows and dangling pointer references that can cause arbitrary data values to be overwritten in memory, and (3) use of un-initialized or incorrectly initialized values (along infrequently executed paths), as these could result in the use of unpredictable values depending on the platform and environment. These are residual errors that are present even in well tested code, and do not manifest in common usage of the program, but in the rare cases that they do, are hard to detect.

3 Related Work

A number of tools have been proposed to find bugs in programs based on static analysis, such as Prefix [5], LCLint [6] and ESC/Java [24]. These tools require the programmer to specify invariants about the program that are then verified by static techniques such as theorem-proving or symbolic execution.

Tools have also been proposed to automatically derive invariants for applications. DAIKON [14] keeps track of common invariants and relationships among variables in a program based on the dynamic execution of the program. This information is then presented to the programmer, who then decides if the invariants derived reveal bugs (defects) in the application. DIDUCE [1] applies the same underlying technique as DAIKON for long-running applications but in a more automated fashion. Engler et al. [4] find commonly occurring patterns in the source code of an operating system kernel and report deviations in these patterns as bugs. While these techniques have been effectively used in finding software defects, they cannot be applied to runtime error-detection in applications.

C-Cured [7] is a system to protect programs from memory errors using a combination of static checking and runtime assertions wherever static checks fail. While C-cured is effective in detecting software errors that violate memory safety, it cannot detect runtime errors resulting from hardware transient faults or software errors that do not violate memory safety.

The only general way to detect runtime-errors is for the programmer to put assertions in the code, as demonstrated by [2]. Saib [3] shows how assertions can increase the reliability of a software program. Zenha-Rela et al. [20] evaluate the coverage provided by programmer-specified assertions in combination with generic techniques such as control-flow checking [27] and Algorithm Based Fault-Tolerance [25]. They find that assertions can significantly complement the coverage provided by generic fault-tolerance techniques.

Voas and Miller [8] provide a general methodology for inserting assertions in programs to maximize error-detection coverage. This method uses programmer knowledge of the application combined with fault-injection to guide the assertion derivation and placement processes. Hiller et al. [18] propose generic classes of detectors for embedded applications but require programmer intervention to choose the correct class of detector for each location. Further, the placement of these detectors is based on fault-injection into the application [19]. Maxion and Tan [26] characterize the space of anomaly-based error detectors and provide a theoretical formulation to benchmark error detectors.

4 Error Detector Format

In this paper, an *error detector* is an assertion based on the value of a single variable of the program at a specific location in its static code. A detector for a variable is placed immediately *after* the instruction that writes to the variable. Since a detector is placed in the static code, it is invoked each time the program location at which the detector is placed is executed.

Consider the sample code fragment in Table 1. 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.

In the code sample, variable *k* is initialized at the beginning of the loop and incremented by 1 within the loop. Within the loop, the value of *k* is dependent 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 in the form: $(k_i - k_{i-1} == 1)$ *or* $(k_i == 0)$.

Table 1: Example code fragment

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

Construction of Detectors. By the above example, a generated set of assertions can be constructed for a target variable by observing the dynamic evolution of the variable over time. A 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, as shown in Table 2.

These rule classes are broadly based on common observations about the behavior of variables in the program. Note that, in all cases, the detector involves only the values of the variable in the current invocation and/or the previous invocation of the detector.

The exception condition involves equality constraints on the current and previous values of the variable, as well as logical combinations (such as and, or) of two of these constraints. The equality constraints take the following forms: (1) $a_i == d$, where d is a constant parameter; (2) $a_{i-1} == d$, where d is a constant parameter; and (3) $a_i == a_{i-1}$. From these three exception constraints, eight unique exception conditions can be formed that 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.

To summarize, for the example involving the loop index variable k, discussed at the beginning of this section, the rule class is Constant-Difference of 1, and the exception condition is ($k_i == 0$).

Table 2: Generic rule classes and their descriptions

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	$((a_i == x$ and $a_{i-1}== y))$ or $(a_i == y$ and $a_{i-1}== x)$	The value of the variable in the current and previous invocations of the detector varies between parameters *x* and *y* alternately.
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	$a_i \in \{x, y, \ldots\}$	The value of the variable in the current invocation of the detector is one of *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.*

5 Dynamic Derivation of Detectors

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

1. The program points at which detectors are placed (both variables and locations) are chosen based on the Dynamic Dependence Graph (DDG) of the program as shown in [9].
2. The program is instrumented to record the run-time evolution of the values of detector variables at their respective locations, and executed over multiple inputs to obtain dynamic-traces of the checked values.
3. The dynamic traces of the checked values obtained are analyzed to choose a set of detectors (both rule class and exception condition) that matches 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* of the detector.

5.1 Detector Tightness and Execution Cost

A qualitative notion of *tightness* of a detector was first introduced in [8]. However, we define tightness in a precise, mathematical sense as the probability that a detector detects an erroneous value of the variable it checks. In mathematical terms, the tightness is the probability that the detector detects an error, given that there is an error in the value of the variable that it checks. The *coverage* of the detector, on the other hand, is the probability that the detector detects an error given that there is an error in any value used in the program. Hence the coverage also depends on the probability of an error to propagate to the detector variable.

To characterize the tightness of a detector, we need to consider both the rule and the exception condition. The tightness also depends on the parameters of the detector and the distribution of the observed correct stream of data values. For a detector to allow an incorrect value to go undetected, either the rule or the exception condition or both must evaluate to true for the value. There are a total of four mutually exclusive cases in which the original and erroneous values each belong to the rule and exception condition, respectively (see Table 3).

Table 3: Probability values for computing tightness

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

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

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

where P(R) is the probability of the value belonging to the rule, and the P(X) is the probability of the value belonging to the exception condition.

The computation of tightness can be automated, since there are only a limited number of rule-exception pairs. There are six types of rule classes and eight types of exception conditions, leading to a total of 48 rule-exception pairs. These can be pre-computed based on the parameters of the detector as well as on the frequency of elements in the observed data stream. The following example illustrates how tightness is computed for a detector.

Example. Consider a detector in which the rule belongs to the class *Bounded-Range* with parameters *min = 5* and *max = 100* and that the exception condition is of the form $(a_i{=}{=}0)$. We assume that the distribution of errors in the detector variable is uniform across the range of all possible values the variable can take (say, N). We also assume that the errors are independent, that an error in the current value of the variable is not affected by an error in the previous value of the variable, and that errors in one detector location are independent of errors in another detector location. These are optimistic assumptions, and the tightness is an upper bound on the coverage of the detector.

Table 4 shows the computed probability values for this detector in terms of N and other parameters. Substituting these in equation (1), we find:

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

as $P(R) + P(X) = 1$ { since the value must satisfy either the rule or the exception condition }

Now, assume that instead of the rule being of the *BoundedRange* class, it belongs to the *Constant* class (with parameter 5). Let us assume that the exception condition is the same as before. For this detector,

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

Substituting in equation (1), yields

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

Note that the probability of a missed error in the first detector is 96 times more than the probability of a missed error in the second detector. Hence, the tightness of the first detector is correspondingly much less than the tightness of the second detector (as expected based on intuition).

The above model is used only to compare the relative tightness of the detectors, and not to compute the actual probabilities (which may be very small). Also, though the tightness of the detector is expressed in terms of *N,* it gets eliminated in the comparison among different detectors for the same variable/location.

Execution Cost. The execution cost of a detector is the amortized additional computation involved in invoking 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 that must be executed in order for a correct value to be validated by the detector. An operation usually corresponds to a single assembly language instruction. Note that the execution cost is computed in the case when the value is correct.

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

Symbol	Probability 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.
$P(X \mid R)$	$(1/N)$	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.

5.2 *Detector Derivation Algorithm*

To derive the detector, the rule class corresponding to the detector is chosen and the associated exception condition is formed. The algorithm to derive a detector for a particular variable and location is given below. We refer to the evolution of a program variable over time as the *stream of values* for that variable.

1. To derive the rule, the rule classes in Table 2 are each tried in sequence against the observed value stream to find which of the rule classes satisfy the observed values. The parameters of the rule are learned based on appropriate samples (for each rule class) from the observed stream. For each rule class, multiple rules are generated depending on the parameters learned.
2. For each rule derived, the associated exception condition is derived based on the values in the stream that do not satisfy the rule. Each of the values that do not satisfy the rule is used as a seed for generating exception conditions for that rule. If it is not possible to learn an exception condition for the observed value, the current rule is discarded and the next rule is tried in the set of rules derived.
3. For each rule-exception pair generated, the tightness and execution cost of the detector is calculated. The detector with the maximum tightness to execution cost ratio is chosen as the final detector for that location. The entire procedure is repeated for each detector location.

6 Coverage of Derived Detectors

This section describes the coverage of the derived detectors using fault-injection experiments. These results are independent of the actual implementation of the detectors (whether in hardware or in software).

6.1 *Experimental Set-up and Workload*

The applications used to evaluate the detectors are the Siemens suite of programs [21]. The Siemens programs considered are *replace* (which performs pattern matching and replacement) *schedule, schedule2 (*which are priority schedulers), *print_tokens*, *print_tokens2 (*which perform lexical analysis) and *tot_info*, (which computes statistics over input data). These are C programs consisting of a few hundred lines of C code and are each equipped with extensive test suites which are used to derive the dynamic detectors.

The detector derivation and fault-injection experiments were done using a modified version of the Simple-scalar simulator [12]. The simulator allows fine-grained tracing of the application and studying its behavior under faults such as hangs, crashes, fail silence violations. We modified the simulator to map the outcome of the simulated program to the real world, as explained in [9]. The experiment is divided into four parts as follows:

1) **Placement of detectors and instrumentation of code.** The dynamic instruction trace of the program is obtained and the Dynamic Dependence Graph (DDG) is constructed from the trace. The points at which detectors (both variables and locations) must be placed are chosen based on our previous work [9]. For each application, up to 100 detector points are chosen by the analysis, which correspond 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 5. The training set consists of 200 inputs, which are randomly sampled from a test suite consisting of 1000 inputs for each program.

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 to type of data violation. The values recorded at the detector locations are then checked by the derived detectors to assess coverage. The coverage of a detector is expressed in terms of the type of program failure it detects i.e. a detector is said to detect a program crash if the program would have crashed had the detector not detected the error. Each application is executed over 10 inputs chosen 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 in the one of these locations. For each application-input combination, 5000 fault-injection runs are performed (5 runs per location).

4) **Computation of false positives.** The application code instrumented with the derived detectors is executed for all 1000 inputs, including the 200 inputs that were used for training. No faults are injected in these runs. If any one of the derived detectors detects an error, then that input is considered to be a false positive as the detectors flag an error even when there was no (injected) error.

6.2 Coverage versus Number of Detectors

The coverage of the detectors derived using the algorithm in Section 5 is evaluated using fault-injections. Figure 2, Figure 3 and Figure 4 show the coverage for crashes, fail-silence violations (fsv) and hangs obtained for the target applications as a function of the number of detectors placed in the application.

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. However, the error coverage of the derived detectors depends on the type of failure and for 100 detectors placed in the code; the coverage obtained for each type of failure is summarized in Table 5.

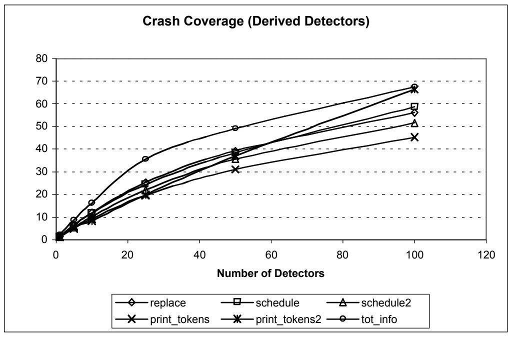

Figure 2: Crash coverage of derived detectors

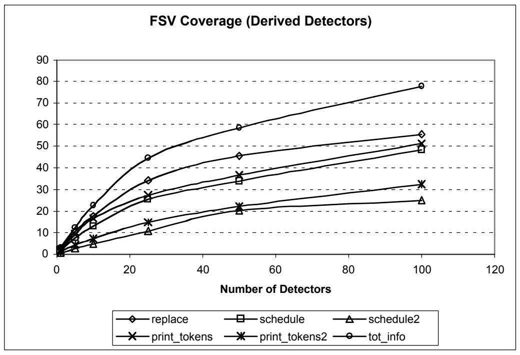

Figure 3: FSV coverage of derived detectors

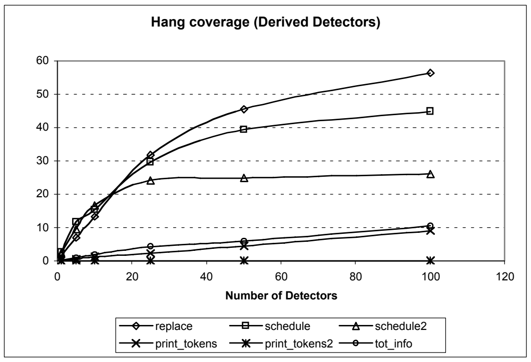

Figure 4: Hang coverage of derived detectors

Table 5: Coverage results for derived detectors

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

Figure 5 shows the percentage of total manifested errors that are detected by the derived detectors. *The derived detectors can detect 50% to 75% of the errors that are manifested in the application.* This is because the majority of errors that manifest in an application are crashes (70-75%) and the rest are fail-silent violations (20-30%) and hangs (0-5%).

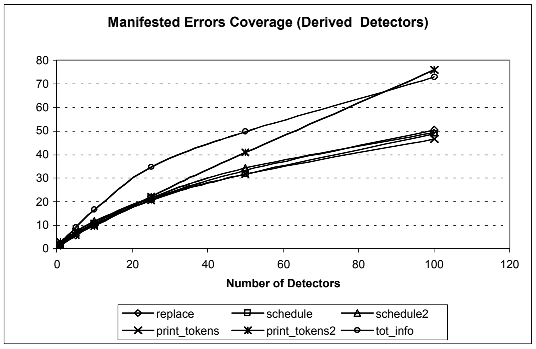

Figure 5: Total error coverage for derived detectors

The results for coverage are for any error that occurs in the data values used by the program, and not just for errors that occur in the detector locations. *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 will detect the error 50-75 % of the time.* Note also that 100 detector locations correspond to less than 5% of program locations in the static assembly code of the benchmark programs.

In comparison, Hiller et al. [18] report a coverage of 80% with 7 assertions for (random) errors that cause failure in their embedded system application. However, in the 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. Furthermore, 7 out of 24 signals for detection in the embedded system considered (about 30% of the system size) are targeted, whereas we place detectors in just 5% of the instructions in the applications considered.

6.3 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 this input do not obey the detector's rule and exception condition learned from the training inputs.

The training set for learning the detectors consists of 200 inputs and the false positives are computed across all 1000 inputs for each application. No faults were injected in these runs. If even a single detector detects an error for a particular input, that input is treated as a false positive.

Figure 6 presents the percentage of false positives for each of the target applications across 1000 inputs. From the figure, the following may be observed:

- For all applications the false positives are no more than 2.5% (with 100 detectors). For the *replace, schedule2, print_tokens* and *print_tokens2* applications, the false positives observed are less than 1%. For the *schedule* and *tot_info* application, the false positive rate is around 2%.
- 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 suggests that inserting more detectors in the application code can increase coverage without increasing the percentage of false positives.

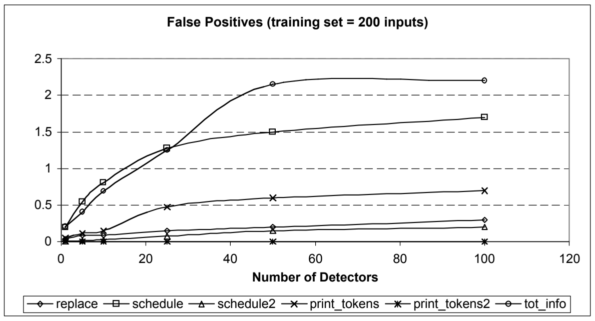

Figure 6: Percentage of false positives for 1000 inputs of each application

When a detector raises an alarm, we need to determine that an error was really present. If the error was caused by a transient fault (focus of this paper), then it is likely to be wiped out when the program is re-executed [22]. If on the other hand, the error was a false positive and hence, a characteristic of the input given to the program, it will be present in the re-executed version of the program, and the detector will raise an alarm again. In this case, the alarm can be ignored, and the program can 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 overhead of re-execution is small. It is possible to reduce the overhead further by using fine-grained, processor-level checkpointing and restarting scheme similar to the one proposed in [11].

6.4 Effect of Training Set Size

- The results reported so far were for coverage and false positives of the derived detectors using training set of 200 inputs from a total of 1000 inputs for each benchmark 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 is fixed at 100 and the error-detection coverage and false positives are evaluated for each application.

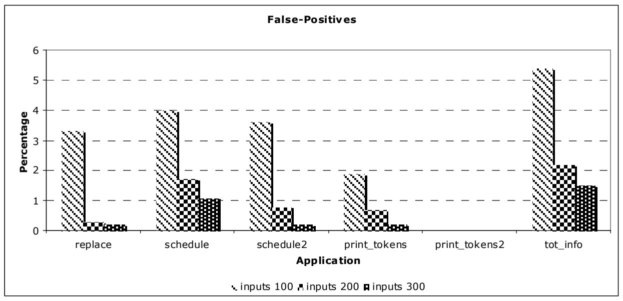

Figure 7: Effect of training set size on false positives

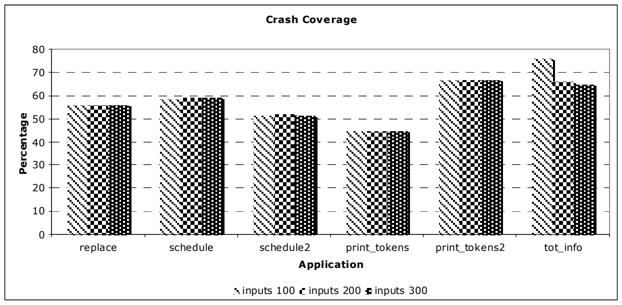

Figure 8: Effect of training set size on crash coverage

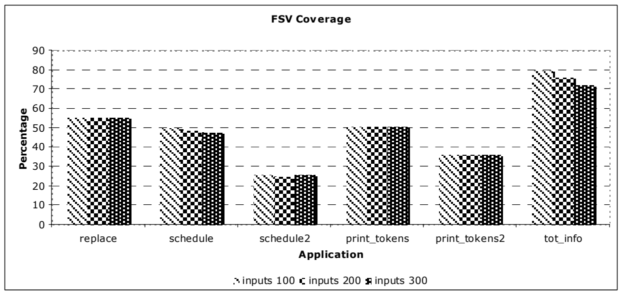

Figure 9: Effect of training set size on FSV

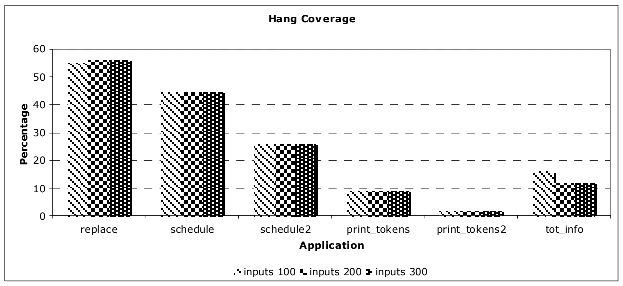

Figure 10: Effect of training set size on hang coverage

The results are shown in Figures 7, 8, 9 and 10 and are summarized as follows:

- 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, for all applications, except tot_info (1.5%.).
- The coverage for crashes and hangs remain constant as the training set size increases (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 is less than 2% for all benchmarks except tot_info (5%).

Thus, increasing the training set size from 100 to 200 decreases the false positives significantly, while increasing it from 200 to 300 does not have as large an impact on false positives. The impact on coverage from increasing the training set size is minimal. Hence, in this paper we choose a training set size of 200, which corresponds to 20% of the inputs used for each program.

7 Hardware Implementation of Detectors

The output of the algorithm to derive detectors is a list of detectors and their associated parameters (see Section 5). This list is used as an input to synthesize hardware modules which implement the detectors. The hardware implementation of error detectors chosen in the design stage encompasses two steps: (i) instrumentation of the target software application with special instructions to invoke the hardware checkers, and (ii) generation of the Error Detector Module (*EDM*), a piece of customized hardware to check at run-time the execution of the program, and flag a signal when one of the detectors fires. These two phases are carried out at compile time, before the application is executed, but can also be executed at application load time. Given the application code (in an intermediate representation, such as assembly code), an automated design flow delivers the instrumented application code and the hardware description of the Error Detector Module tailored for the target application. The *target processor description* (a DLX-like processor in the current implementation [23]) and the *configuration information* are used to extract (from the main pipeline of the processor) the signals that are needed by the hardware Error Detector Module.

In this paper, we discuss the hardware implementation of the Error Detector Module in context of the Reliability and Security Engine (RSE) framework [13]. The RSE is a reconfigurable processor-level framework that can provide a variety of reliability features according to the needs and constraints imposed by the user or the application [13]. The RSE Framework hosts (1) *RSE modules*, such as the Error Detector module, providing reliability and security services and (2) the *RSE Interface,* which provides a standard, well defined, and extendible interface between the modules and the main processor pipeline. The interface collects the intermediate pipeline signals and converts it to the format required by the hardware modules. The application interfaces with the modules (and hence to the Error Detector Module) using special instructions called CHECK instructions.

Each detector in the *list of detectors* derived in the design phases is characterized by the following attributes: (1) location in terms of the Program Counter value, (2) processor registers to check, and (3) detector class and exception parameters. Multiple CHECK instructions are used to load the specific parameters (i.e., rule class and exception condition) for the detectors into hardware and enable/disable the module.

7.1 Detailed Description of the EDM

In this section, we describe the overall architecture of the Error Detector Module (EDM) referring to Figure 11. We assume that the required signals are provided through an interface to the processor similar to the RSE interface described in 7. The components in the Error Detector Module are described below:

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}, whereas a is a generic register). This component delivers the required values a_i and a_{i-1} when a detector is executed as required by the expressions reported in Table 1. Note that only the values of registers checked by any detector have to be stored in the SRF. When a new value *regValue* is written at time *i* by the processor in the register *R* of the processor file (pointed by the value *regSel*), a copy of the new value R_i is stored in the SRF, keeping also the value R_{i-1}. Since not all registers of the processor architecture have to be checked by 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 the block *Phys2Log*.

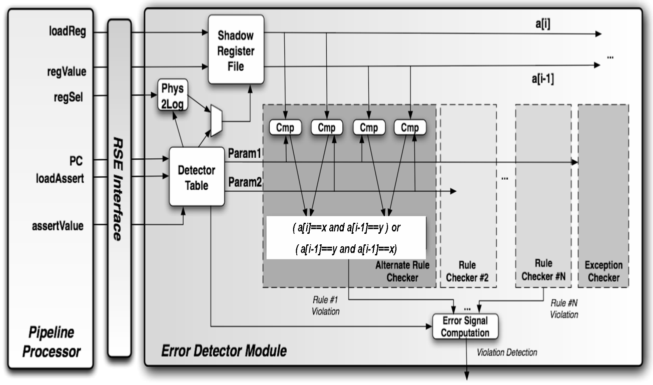

Figure 11: Architectural diagram of synthesized processor with detection modules

Detector Table stores the information needed for a detector. The Detector Table grows with the number of detectors needed by an application. It is implemented by the following component: (1) comparators checking the current PC against the PCs triggering the active detectors; (2) a RAM hosting the parameters of rules and exceptions. When a detector is triggered by the current PC, the Detector Table selects (1) the register R that has to be checked from the SRF forcing the values Ri-1 and Ri-1 to be placed on the 2 data-path busses, and (2) activates the Rule and Exception Checkers to compute the detector conditions, and the Error Signal Computation flags the Violation Detection signal to indicate a detected malfunctioning.

Rule and Exception Checkers are the 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 set of checkers instantiated is equal to the number of detector classes and not to the number of detectors for an application.

7.2 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 type 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*), memory blocks that can store up to 4096 bits. Four LUTs and four FFs form a logic unit called a *Slice*.

Area and Clock Period Overhead. Table 6 reports the synthesis results in terms of area (i.e., FFs, LUTs, BRAM and total Slices) and minimum clock frequency, for the reference DLX processor and the complete RSE Interface. The area overhead (with respect to the single superscalar DLX processor) of the single EDM is about 30%, while the area overhead of the complete (including the RSE Interface EDM) is about 45%. We also find that the increase in clock period time is about 5% with the EDM and the RSE combined

Table 6: Area and timing overheads

	FFs	LUTs	BRAMs	Slices	Clk Period [ns]
DLX Processor	4873	16395	0	9526	58.8
RSE Interface	2465	2329	0	1420	2.01

8 Conclusions and Future Work

This paper proposes a novel technique for preventing a wide range of data errors from corrupting the execution of a generic 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 are derived to maximize the error detection coverage for a target application. The paper also presents an automatic framework for synthesizing the detectors in hardware to enable low-overhead run-time checking of the application execution. The coverage of the derived detectors is evaluated using fault-injection, and the hardware implementation of the detectors is synthesized to obtain area and performance overheads.

Future work will involve building more comprehensive detector classes and using source-level information from the programs to derive detectors

Acknowledgments

This work was supported in part by the US Office of Naval Research, the Defense Advanced Research Projects Agency (MURI Grant N00014-01-1-0576), the Gigascale Systems Research Center (GSRC/MARCO), NSF grants CNS-0406351, and CNS-05-24695.

References

1. S.Hangal and M. Lam, Tracking down software bugs using automatic anomaly detection, *Proc. 24th International Conference on Software Engineering (ICSE)*, pp. 291-301, 2002.
2. Andrews, D., Using executable assertions for testing and fault tolerance, *Proc. 9th Fault-Tolerant Computing Symposium (FTCS),* pp. 20-22, 1979.
3. Saib S.H., Executable assertions: An aid to reliable software, Proc. *11th Asilomar Conference on Circuits, Systems, and Computers*, pp. 227-281, 1978.
4. D. Engler, D.Y. Chen, S. Hallem, A. Chou,, B. Chelf., Bugs as deviant behavior: A general approach to inferring errors in system code, *Proc, Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pp 57-72, 2001.
5. W. R. Bush, J. D. Pincus and D. J. Sielaff., A static analyzer for finding dynamic programming errors, *Software: Practice and Experience,* 30(7), pp. 775-802, 2000.
6. D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In Proc. *ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87-96, December 1994
7. J. Condit, M. Harren, S. McPeak, G. Necula, and W. Weimer. CCured in the real world. In A*CM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 232--244, June 2003.
8. J. Voas and K. Miller, Putting assertions in their place, in Proc. of the *International Symposium on Software Reliability Engineering (ISSRE),* November, 1994.
9. K.Pattabiraman, Z.Kalbarczyk, and R.K. Iyer, Application-based metrics for strategic placement of detectors Proc. 11th *International Symposium on Pacific Rim Dependable Computing (PRDC)*, pp. 75-82, December, 2005
10. S. Chandra, and P. Chen. How Fail-Stop are Faulty Programs?, *Proc. 28th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pp. 240-249, June 1998..
11. N. J. Wang and S. J. Patel, ReStore: Symptom-based soft error detection in microprocessors, *Proc. International Conference on Dependable Systems and Networks*, pages 30-39, June 2005.
12. Doug C. Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors the simplescalar tool set., *Technical Report 1308, Computer Sciences Department,* University of Wisconsin--Madison, July 1996.
13. N. Nakka, J.Xu, Z.Kalbarczyk, R.K. Iyer., An architectural framework for providing reliability and security support, *Proc. Intl. Conference on Dependable Systems and Networks (DSN)*, pages: 585-594, 2004.
14. Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering,* 27(2):1-25, 2001
15. C. Basile, L. Wang, Z. Kalbarczyk and R.K Iyer., Group communication protocols under errors, *Symposium on Reliable Distributed Systems (SRDS)*, pp 35, 2003.
16. S. Chandra and P. M. Chen. Whither Generic Recovery from Application Faults? A Fault Study Using Open Source Software. *Proc. International Conference on Dependable Systems and Networks (DSN)*, pages 97-106, 2000.
17. W. Gu, Z. Kalbarczyk, R.K. Iyer, Z. Yang, Characterization of Linux Kernel Behavior under Errors, *Proc. International Conference on Dependable Systems and Networks (DSN'03)*, pp. 459-468, June 2003.
18. M. Hiller, Executable detectors for detecting data errors in embedded control systems, *Proc. International Conference on Dependable Systems and Networks (DSN*), pp. 24-33, 2000.
19. M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. In Proc. *Intlernational Conference on Dependable Systems and Networks (DSN)*, pages 135-144, 2002.
20. M. Zenha Rela, H. Madeira, J.G. Silva, Experimental evaluation of the fail-silent behavior in programs with consistency checks, *Proc. 26th Fault-Tolerance Computing Symposium, (FTCS)*, pp. 415-424, 1996.
21. M. Hutchins, H. Foster, T. Goradia, T. Ostrand, Experiments of the effectiveness of dataflow- and control-flow-based test adequacy criteria, *Proc. International Conference of Software Engineering (ICSE)*, pp. 191-200, 1994.
22. I. Lee and R. K. Iyer, Software Dependability in the Tandem GUARDIAN System, *IEEE Trans. on Software Engineering,* Vol. 21, No. 5, pp. 455-467, May 1995.
23. J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publ., 1996.
24. C. Flanagan, K. R. M. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java, In P*roc. ACM SIGPLAN Conference on. Programming Languages Design and Implementation (PLDI)*, pages 234-245, 2002.
25. K. Huang, J. Abraham, Algorithm-based fault tolerance for matrix operations, *IEEE Trans. on Computers,* C-33, pp. 518-528, 1984.
26. R.A.Maxion, M.C. Tan, Anomaly detection in embedded systems, *IEEE Trans. on Computers*, 51(2), pages 108-120, 2002
27. Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, and J.A. Abraham, Design and evaluation of system-level checks for on-line control flow error detection, *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 627-641, June 1999.