Automated Derivation and Hardware Implementation of Application-Specific Error Detectors

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
1308 W. Main Street, Urbana, IL 61801
{pattabir, saggesse, dchen8, kalbar, iyer}@crhc .uiuc.edu

Abstract - *This paper proposes a novel technique for automated derivation of fine-grained, application-specific error detectors. An algorithm based on dynamic traces of application execution is developed for extracting the optimal set of error detectors for a target application. An automatic framework is proposed for synthesizing the derived detectors in hardware and enabling low-overhead run-time checking of the application execution. Coverage (evaluated using fault injection) of the error detectors obtained using the proposed methodology, the additional hardware resources, and performance overhead for several benchmark programs are also reported.*

1. Introduction

This paper presents a technique to derive and implement error detectors to protect an application from *data errors*. Typically, many errors in a program do not manifest in the program's outcome [1]. If they manifest, they cause a divergence in data values observed during the error-free execution of the program. We refer to these errors as data errors. Data errors may cause the program to crash, hang or produce incorrect output (fail-silent violations). These errors can result from incorrect computation, and would not be caught by traditional techniques such as ECC in memory.

This paper contributes with: (i) a procedure for automated generation of a class of application-specific detectors and their implementation in hardware in the form of concurrent checkers tailored to the application; and (ii) an experimental assessment of the proposed methodology.

The proposed methodology is applied to derive an optimal set of detectors for several benchmark programs. Experimental evaluation of coverage (assessed via fault injection), the additional hardware resources, and performance overhead indicate that: (1) The coverage of the detectors derived ranges in 40-60% for crashes and in 20-60% for fail-silent violations, when 100 detectors are placed in the application code; (2) A hardware implementation of the detectors incurs a performance slowdown of around 5%, with acceptable area and power overhead; (3) False positives (detectors flagging an error when no error occurs) are less than 6% in the vast majority of considered benchmarks

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 processor register file. 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, 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 are usually hard to detect.

2. Approach

The derivation and implementation of the error detectors in hardware encompasses four main phases as depicted in Figure 1. The analysis and design phases are related to the derivation of the detectors, while the synthesis and checking phase are related to the implementation and use of the detectors at run-time.

During the *analysis* phase, the locations and variables for placing detectors to maximize coverage are identified, based on the execution of the code and the Dynamic Dependence Graph of the program. This phase is based on the technique we proposed in [10]. 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 choose the best detector that matches the observed values for the variable, based on a set of pre-determined generic detector classes.

After this stage, the detectors can be integrated into application code as assertions or implemented in hardware. The *synthesis* phase converts the assertions generated to a HDL description that is synthesized in hardware. It also inserts special instructions in the application code to invoke and configure the hardware detectors. Finally, during the *checking* phase, the custom hardware detectors are deployed in the system to provide low-overhead run-time error detection.

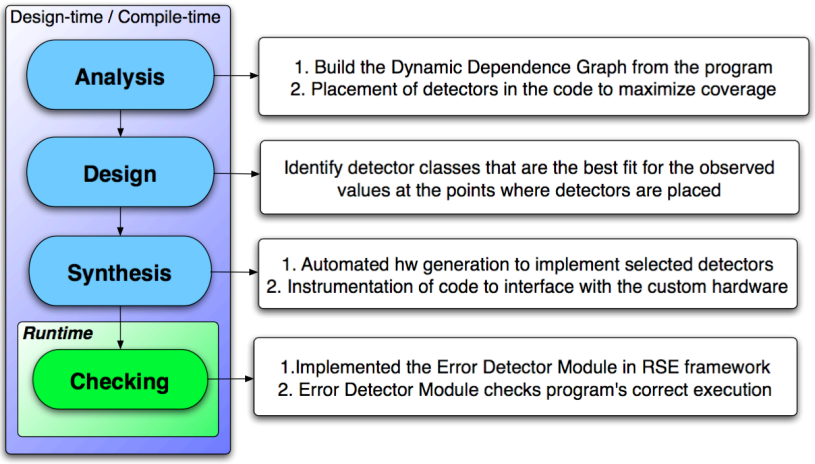

Figure 1: Steps in detector derivation and implementation process

3. Related Work

Broadly, error detectors can be classified based on two criteria: (1) how the detectors are derived (static or dynamic) and (2) how the checking is performed (static or dynamic). Tools such as PREFIX [5], Bandera [6], ESC/JAVA [7] and C-Cured [8] analyze the application code and remove errors before the application is executed. These fall in the category of statically derived, statically-checked detectors.

DAIKON [13] infers invariants about the code from the dynamic execution of the program for a characteristic test suite. The invariants are statically checked by the programmer or by an automatic theorem-proving tool to eliminate false or spurious invariants that are a property of the test-suite rather than the code. The main difference between DAIKON and our approach is that DAIKON derives detectors comprising of multiple program variables at an instant of time, whereas our approach derives detectors based on the evolution of a single-program variable over time.

Dynamically-checked detectors can be derived statically from the specifications or from code properties. Examples of this class of detectors correspond to executable assertions embedded by the programmer in the code. These detectors can also be derived using automated tools that check for violations of specifications or code properties. For example, the Purify [3] and Valgrind [4] tools dynamically check for memory leaks and memory corruptions, while the ERASER[9] tool dynamically checks for data races in a multi-threaded application. The main difference between these detectors and our detectors are that our detectors are derived dynamically based on the application properties, and do not require specifications or programmer intervention.

The technique proposed in this paper falls in the category of dynamically-derived, dynamically-checked detectors. Another example in this category is the DIDUCE tool [2]. The main difference between the detection performed by DIDUCE and our detectors is that DIDUCE is used for detecting errors in programs for debugging purposes whereas our detectors are aimed at detecting random errors in production settings.

4. Detector Derivation

In this study, an *error detector* is defined as a check on the value of a single variable of the program at a specific location in the static code of the program. Thus, a detector is essentially a software assertion. A detector for a variable is placed immediately *after* the instruction that writes to the variable. A detector is placed in the static code of the program and is invoked each time the program location at which the detector is placed is executed.

Class Name	Generic Rule (a_i , a_{i-1})	Description
Constant	$(a_i == c)$	The value is always a constant, which is given by parameter *c*
Alternate	$(a_i == x \text{ and } a_{i-1} == y)$ or $(a_i == y \text{ and } a_{i-1} == x)$	The value varies between parameters *x* and *y* alternately
Constant-Difference	$(a_i - a_{i-1} == c)$	The value differs from its previous value by a constant parameter *c*
Bounded-Difference	$(min <= a_i - a_{i-1} <= max)$	The difference between the previous and current values lies between parameters *min* and *max*
Multi-Value	$a_i \epsilon \{ x, y, \ldots \}$	The value is one of *x, y* etc., which are the parameters of the rule
Bounded-Range	$(min <= a_i <= max)$	The value lies between the parameters, *min* and *max*

Table 1: Generic Rule Classes and their Descriptions

In our current implementation, the detector involves only functions of the current and previous values of the selected variable at the detector's location. We refer to the current value of the detector variable a as a_i and the previous value as a_{i-1}. A generated set of detectors can be constructed for a target variable by observing the evolution of the variable over time. A detector consists of a *rule* describing the allowed values of the selected 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. The detector rules can belong to one of the six generic classes and are parameterized for the variable checked. Table 1 shows the set of rule classes that are used to derive detectors.

The exception condition only 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) $a_{i-1} == e$, where e is a constant parameter; and (3) $a_i == a_{i-1}$.

The detectors are automatically derived based on the dynamic trace of values produced during the application's execution. 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 [10]. The program is then 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. These traces are then analyzed to choose a set of detectors (both rule class and exception condition) that matches the observed values. A probabilistic model for coverage is then applied to the set of chosen detectors to find the best detector for that location (not discussed here due to space constraints).

In order 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. In order to derive the rule, the rule classes in Table 1 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. The set of all rule classes is considered in step 2.
2. For each rule derived in step 1, 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. The exception conditions generated are based on the equality constraints described before and logical combinations of two of these constraints. 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 in step 1. The set of all rule-exception pairs generated is considered in step 3.
3. For each rule-exception pair generated, the best detector is chosen according to a probability model that estimates coverage, for that location. The entire procedure is repeated for each detector location.

The output of the algorithm is a list of detectors that are used to synthesize the hardware modules in Section 5.

5. Hardware Implementation

The hardware implementation of error detectors derived in the design stage encompasses two steps: (i) instrumentation with the CHECK instructions[1] of the target software application, 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. Given the application code (in an intermediate representation, such as assembly code) the design flow delivers – in an automatic fashion without the designer involved in any design decision – the instrumented application code and the hardware description of the Error Detector module tailored for the target application.

The technique we propose is general and can be adapted to any processor. The information required from the main pipeline (e.g., the value of the PC and state of internal registers) can be found in any modern processor. In this paper, we discuss the hardware implementation of the Error Detector Module in context of the Reliability and Security Engine (RSE) framework[12] and of a DLX-like processor [1]. 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

In the following we describe the overall architecture of the Error Detector Module referring to Figure 2. We assume that the required signals are provided through an interface to the processor similar to the RSE interface. There are several components in the Error Detector Module described below: *Shadow Register File* (SRF) keeps track of current and last values of the microprocessor's registers checked by the detectors. *Detector Table* stores the information needed for a detector. *Rule and Exception Checkers* – are the actual data-paths used to carry out the computation of the detector rules and exceptions. *Violation Detector* – uses the results of the rule and exception checkers to flag an error, indicating a malfunctioning when both the clause and the exception fail.

[1] These are special instructions that are used to invoke the Error-Detector Module from the target application.

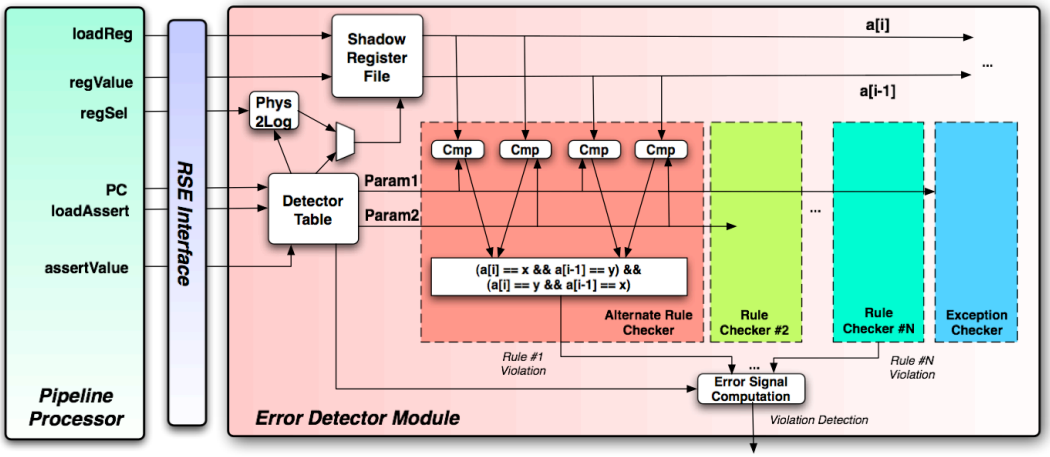

Figure 2 – Detailed Architectural View of the Error Detector Module

6. Experimental Evaluation

This section includes several experimental results of the proposed technique: (1) assessment of different detector sets in terms of their ability to detect crashes, hangs and fail-silent violations; and (2) the area and performance overheads when the detectors are implemented in hardware.

6.1 Detector Derivation and their Coverage

In order to perform the detector derivation and fault-injection experiments, a modified version of the Simple-scalar simulator [11] is used. The simulator allows fine-grained tracing of the application and studying its behavior under faults such as hangs, crashes, fail-silent violations. The results from the simulator represent the behavior of the processor augmented with the hardware *Error Detection Module*. The experiment is divided into three parts: (1) Placement of detectors and instrumentation of code; (2) Deriving the detectors based on training set; and (3) Fault-injections and coverage estimation.

In the *first* part, 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 [10]. For each application, up to 100 detector points are chosen by the analysis. In the *second* phase, the detectors are derived. The simulator records the values of the selected variables at the detector locations for representative inputs. The dynamic values obtained are used in the learning phase to derive the detectors using nearly 100 inputs in the training set. Finally the *third* phase consists of fault-injection experiments performed by flipping randomly chosen single bits in data-values produced during the course of a 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 into crash, hang, success or fail-silent data violation. The values recorded at the detector locations are then checked offline by the derived detectors to assess their coverage.

The applications used to evaluate the detectors are the Siemens suite [14] of programs. These are C programs consisting of few hundred lines of C code. Each application is executed over 10 new inputs (unseen during the learning phase) and for each input 1000 random locations are chosen for fault-injections. For each location, five random bits are corrupted (one at a time), leading to a total of 5000 fault-injections for each application/input combination.

Figure 3, Figure 4 and Figure 5 show the coverage for crashes, hangs and fail-silent violations (fsv) obtained for the target applications as a function of the number of detectors placed in the application. The main results are:

- While coverage for all three classes of failures (crash, hang and fail-silent violations) increases as the number of detectors increases, there is a significant overlap in the errors detected by different detectors which leads to a plateau effect in the coverage;
- Error coverage varies significantly across applications depending on the type of failure. For 100 detectors placed in the code, coverage for crash failures varies between 45% (*print_tokens)* and 60% (*schedule*), for hangs between 2% (*print_tokens2*) to 40% (*schedule*) and for fsv, from 20% (*schedule2*) to 60% *(tot_info*).

False-positives can occur when a detector flags an error even if there is no error in the application. Some of the detectors may fail on some of the inputs as the values at the detector points for these inputs may not obey the detector's rule or exception condition learned from the training inputs. Figure 6 presents the percentage of false-positives for each of the target applications across 1000 inputs. If even a single detector detects an error for a particular input, that input is treated as a false-positive. For all applications except *tot_info,* the false-positives observed are less than 6% (for 100 detectors). For *the tot_info* application, the observed false-positives are 16%.

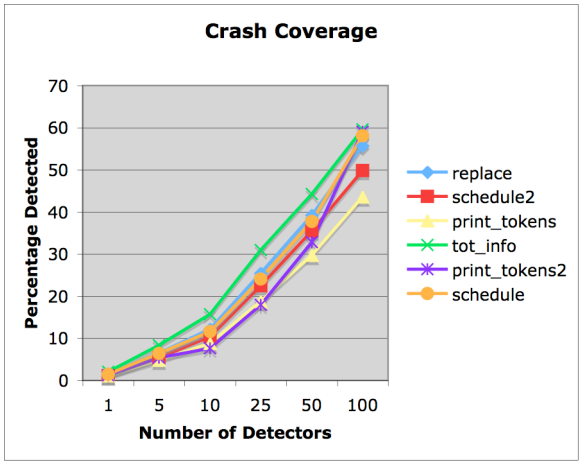

Figure 3: Crash Coverage for actual detectors

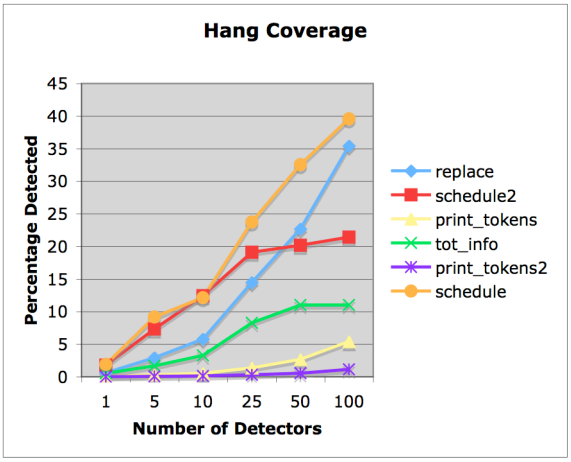

Figure 4: Hang Coverage for actual detectors

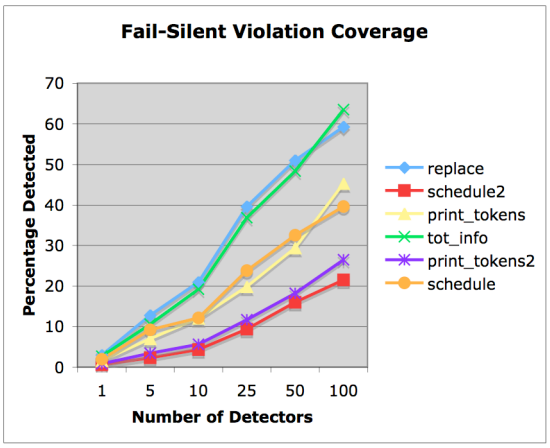

Figure 5: FSV coverage for inserted detectors

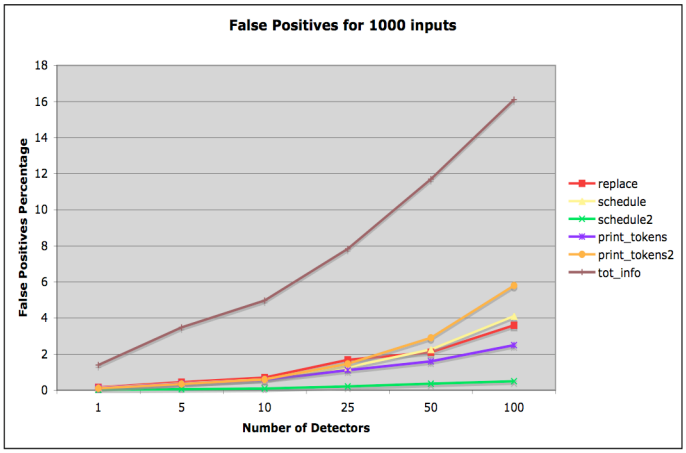

Figure 6: Percentage of False-Positives for 1000 inputs of each application

6.2 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*), which are memory blocks that can store up to 4096 bits. Four LUTs and four FFs compose a logic unit called *Slice*.

Table 2 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. Table 3 gives the synthesis results in terms of area and minimum clock period for different configurations of the Error Detector module for the workloads reported in the first column. For different workloads, the number of slices required for the implementation of the Error Detector modules ranges between 2685 and 2915, while the number of additional BRAMs is 9. The last two columns of Table 3 show the area overhead, for the Error Detector module (EDM) and for a complete implementation of the RSE, respectively. The area overhead of the single EDM is about 30%, while the area overhead of the complete implementation is about 45%.

A measure of the performance overhead is given by:

$$(T_{with\ EDM} - T_{without\ EDM}) / T_{without\ EDM}$$
$$= [\ Extra\ Clock\ Cycles * (T_{CK,\ with\ ED} - T_{CK,\ without\ EDM})\ ] / (\ Total\ Clock\ Cycles * T_{CK,\ without\ EDM})$$

where $T_{with\ EDM}$ and $T_{without\ EDM}$ are the total execution times with and without Error Detector module respectively, *Extra clock cycles* is the number of additional clock cycles required to execute the code with the CHECK instructions, $T_{CK\ with\ ED}$ and $T_{CK\ without\ ED}$ are the minimum clock period of the overall system with and without the Error Detector module, respectively.

Due to space constraints, we do not report the results for all the workloads, but we report only the workload with the largest time overhead, i.e., *schedule2*. The number of extra clock cycles is 594, while the total number of clock cycles is nearly 1 million, $T_{CK\ with\ ED}$ is 58.82 ns and $T_{CK\ without\ ED}$ is 55.55 ns. From this, we can calculate the total performance overhead to be about 5.6%.

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

Table 2: Area and timing results for the DLX processor and the RSE Framework

Workload Name	Number of 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 3: Area and timing results for Error Detector modules for different workloads.

7. Conclusions and Future Work

This paper has proposed a novel technique for preventing a wide range of data errors from corrupting the execution of a generic application. This technique consists in automated derivation of fine-grained, application-specific error detectors by an algorithm based on dynamic traces of application execution. A set of error detector classes, parameters and locations, are produced in order to maximize the error detection coverage for a target application. The paper also presents an automatic framework for synthesizing the optimal 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, and performance overhead for several benchmark programs are also reported.

Future work will involve (1) Evaluating the technique on larger benchmark programs, (2) Reducing the number of false-positives encountered and (3) Increasing the coverage for fail-silence violations by deriving detectors that are a function of the application's inputs.

Acknowledgements

This work was supported in part by the US office of Naval Research, Defense Advanced Research Projects Agency (MURI Grant N00014-01-1-0576), Gigascale Systems Research Center (GSRC/MARCO), NSF Next-Generation Software (grant number CNS-0406351), and Motorola Corporation.

References

[1] G.P. Saggese et al., Microprocessor Sensitivity to Failures: Control vs Execution, Combinational vs Sequential", Proceedings of DSN Conference 2005.

[2] S.Hangal and M. Lam, Tracking down software bugs using automatic anomaly detection, Intl. Conference on Software Engineering 2002.

[3] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In Proceedings of the USENIX Winter Technical Conference, 1992.

[4] Nethercote, N. and Seward, J. 2003. Valgrind: A program supervision framework. In Proceedings of the 3rd International Workshop on Runtime Verification (RV).

[5] W. Bush et al. A static analyzer for finding dynamic programming errors. Software: Practice and Experience, 30(7), 2000.

[6] Corbett, et. al., Bandera: Extracting finite-state models from Java source code. In Proc. 22nd International Conference on Software Engineering (ICSE), June 2000

[7] Flanagan et. al., Extended static checking for Java, Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation..

[8] J. Condit, et al.. CCured in the real world. In Proceedings of the ACM SIGPLAN 2003.

[9] S. Savage et al., A dynamic data race detector for multi-threaded programs. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, October 1997.

[10] K.Pattabiraman et al., Application-Based Metrics for Strategic Placement of Detectors, To appear in Pacific rim Dependable Computing (PRDC), 2005.

[11] D. Burger, T. Austin, and S. Bennett, Evaluating Future Microprocessors: The SimpleScalar ToolSet, University of Wisconsin-Madison, Computer Sciences Department, Technical Report CS-TR-1308, July 1996.

[12] N. Nakka, et al., An Architectural Framework for Providing Reliability and Security Support, Proc. Intl. Conference on Dependable Systems and Networks (DSN), 2004.

[13] M.D Ernst et. al., Dynamically Discovering Likely Program Invariants to Support Program Evolution, IEEE Transactions on Software Engineering. Volume 27, Issue 2, Feb 2001.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, Experiments of the Effectiveness of Dataflow- and Control-flow Based Test Adequacy Criteria, Proc. Intl. Conference of Software Engineering (ICSE), 1994.