Application-Based Metrics for Strategic Placement of Detectors

Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer
Center for Reliable and High-Performance Computing,
Coordinated Sciences Laboratory,
University of Illinois at Urbana-Champaign
{pattabir, kalbarcz, rkiyer}@uiuc.edu

Abstract

*The goal of this study is to provide low-latency detection and prevent error propagation due to value errors. This paper introduces metrics to guide the strategic placement of detectors and evaluates (using fault injection) the coverage provided by ideal detectors embedded at program locations selected using the computed metrics. The computation is represented in the form of a Dynamic Dependence Graph (DDG), a directed-acyclic graph that captures the dynamic dependencies among the values produced during the course of program execution. The DDG is employed to model error propagation in the program and to derive metrics (e.g., value fanout or lifetime) for detector placement. The coverage of the detectors placed is evaluated using fault injections in real programs, including two large SPEC95 integer benchmarks (*gcc *and* perl*). Results show that a small number of detectors, strategically placed, can achieve a high degree of detection coverage.*

1. Introduction

This paper presents a technique of inserting detectors or checks into programs to prevent/limit fault propagation due to *value errors.* Value errors are errors that cause a divergence from the program values seen during the error-free execution of the application. These errors can lead to application crash, hang, or fail-silent violations (when the program produces an incorrect result). Data from real systems has shown that, while many crashes are benign, severe system failures often result from latent errors that cause undetected error propagation [14][19]. These latent errors can cause file corruption [10], propagate to other processes in a distributed system [2], or result in checkpoint corruption [5] prior to a system crash. It is a common assumption that crashes are benign and that there is a mechanism in a system that ensures that when the program encounters an error (that ultimately leads to a crash), the application will crash instantaneously (crash-failure semantics).

To guarantee crash-failure semantics for a program, we need some form of checking mechanisms in the system. Such support can take many forms, including protection at multiple levels and duplication both in hardware and software. Recent commercial examples of such approaches include: (i) IBM G5, which, at the processor level, employs two fully duplicated lock-step pipelines to enable low-latency detection and rapid recovery [18] and (ii) HP NonStop Himalaya, which, at the system level, employs two processors running the same program in locked step. Faults are detected by comparing the output of the two processors at the external pins on every clock cycle [7]. Although these are very robust solutions, due to their high cost and significant hardware overhead, they are usually used only in high-end mainframes and servers intended for mission-critical applications, not in commercial off-the-shelf (COTS) systems.

This paper introduces metrics to guide strategic placement of detectors and evaluates (using fault injection) the coverage provided by ideal detectors (an ideal detector is one that detects 100% of the errors that are manifested at its location in the program) at program locations selected using the computed metrics. Results show that a *small number of detectors, strategically placed, can achieve a high degree of detection coverage.* The issues of development of actual detectors and the performance implications of embedding the detectors into the application code are not addressed in this study. Examples of potential detectors are consistency checks on the values in the program, such as range-checks and instruction sequence-checks [11]. In this paper:

- The program's code and dynamic execution are analyzed, and an abstract model of the data-dependencies in the program called the Dynamic Dependence Graph (DDG) is built.
- Several metrics such as *fanout* and *lifetime* are derived from the DDG and used to strategically place/embed detectors in the program code to maximize the coverage. A detector's coverage depends on two factors: (i) the effectiveness (coverage) of the placement of the detector, i.e., how many errors manifest at the location where the detector is embedded and (ii) the effectiveness (coverage) of the detector itself, i.e., what fraction of errors manifested at the detector's location are captured.

The key findings from this work are:

- A single detector placed using the *fanout* metric can achieve 50-60% crash-detection coverage for large benchmarks (*gcc* and *perl*).
- A small number of detectors placed using the *lifetime* metric can achieve high coverage for large benchmarks. For example, it is possible to achieve about 80% coverage with 10 detectors and 90% coverage with 25 detectors embedded in the *gcc* benchmark.
- Although the placement of detectors is geared toward providing low-latency detection and preventing propagation by preemptively detecting potential crashes, the placed detectors are also effective at detecting fail-silence violations (i.e., the application terminates normally but produces incorrect results) (30-70%) and hangs (50-60%).

2. Related Work

In recent years, several studies have addressed the issue of strategic placement of detectors in application code. Hiller et al. [12] use Error Propagation Analysis (EPA) to determine where detectors or checks should be inserted in an embedded control system. It is assumed that the checks have ideal coverage (100%) and are inserted at points (signals) at which error detection probability is the highest. Voas [20] proposes the "avalanche paradigm," a technique to place assertions in programs before faults in the program propagate to critical states. Goradia [9] evaluates the sensitivity of data values to errors from a software testing perspective.

DAIKON [8] is a dynamic analysis system for generating likely program invariants to detect software bugs. Narayanan et al. [16] use the loop invariants produced by DAIKON to detect soft errors in the data cache. DAIKON places assertions at the beginnings and ends of loops and procedure calls. However, this may not be sufficient to provide low-latency error detection, as the application/system may misbehave long before the assertion point is reached. Benso et al. [3] present a compiler technique to detect critical values in a program. The criticality of a variable is calculated based on the lifetime of the variable and how many other variables it affects. This technique can protect against faults that originate in the critical variable and propagate to other variables, but it does not protect against faults that are propagated to the critical variable from other locations.

3. Computation Model: Dynamic Dependence Graph (DDG)

The computation is represented in the form of a Dynamic Dependence Graph (DDG), a directed-acyclic graph (DAG) that captures the dynamic dependencies among the values produced in the course of program execution. In this context, a *value* is a dynamic definition (assignment) of a variable or memory location used by the program at runtime. *A value may be read many times, but it is written only once*. If the variable or location is rewritten, it is treated as a new value. Thus, a single variable or memory location may be mapped onto multiple values.

This is similar to the Static Single Assignment (SSA) form used by compilers [1]. In SSA, however, each static assignment of a variable is treated as a new value, while in a DDG each dynamic assignment of a variable is treated as a new value

A node in the DDG represents a value produced in the program, and it is associated with the dynamic instruction that produced the value. In the DDG, edges are drawn between nodes representing the operands of an instruction and nodes representing the value produced by the instruction. The *edge* represents the instruction, the *source node* of the outgoing edge corresponds to an instruction operand, and the *destination node* corresponds to the value produced by the instruction.

Table 1 shows an example code fragment, and Figure 1 shows the DDG corresponding to that code fragment. The code computes the sum of the elements of an array *A* of 5 integers (denoted by *size*) and stores the sum in the variable *sum*. The table shows the mapping between the DDG nodes and the instructions, as well as the effect of executing the instructions. Not all nodes in the DDG correspond to instructions, e.g., nodes 1, 3, 8, 13, 23, and 28 represent memory locations used by the code fragment.

The following observations can be made based on the DDG:

- Every value-producing instruction has a corresponding node in the DDG, shown by an arrow from the instruction to its node label in the DDG.
- Memory locations are represented as DDG nodes when they are first read or written. For example, in Figure 1, nodes 1 and 28 represent memory locations *size* and *sum,* respectively, and nodes 3, 8, 13, 18, and 23 represent the array locations A[0] to A[4].

Table 1: Example code fragment to compute array sum

Code fragment	Explanation	Nodes in DDG
ADDI R1, R0, 0	R1 ← R0	6
LW R2, [size]	R2 ← [size]	2
ADDI R4, R0, 0	R4 ← R0	0
LOOP: LW R3, R1[A]	R3 ← A[R1]	5, 10, 15, 20, 25
ADD R4, R4, R3	R4 ← R4 + R3	4, 9, 14, 19, 24
ADDI R1, R1, 1	R1 ← R1 + 1	6, 11, 16, 21, 26
BNE R1, R2, LOOP	If (R1!=R2) then goto Loop	7, 12, 17, 22, 27
SW [Sum], R4	[Sum] ← R4	28

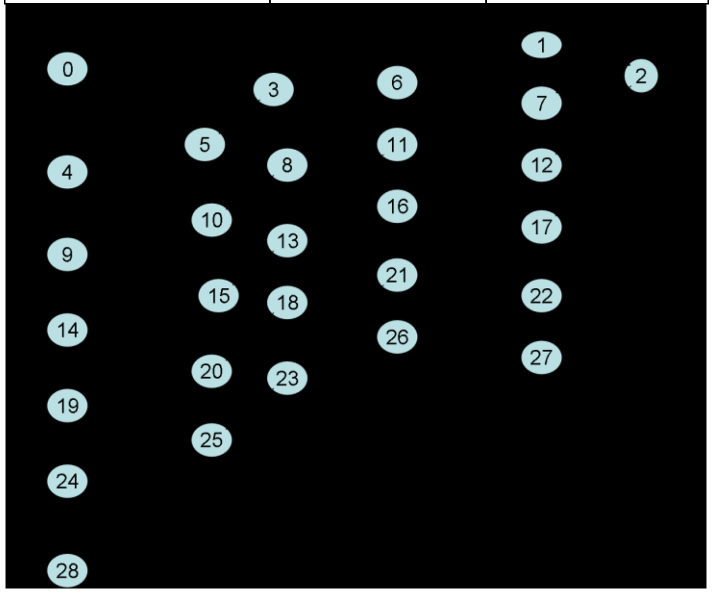

Figure 1: DDG corresponding to code fragment in Table 1

- Constants are not represented in the DDG (e.g., 0 and 1 are not represented, though they appear as instruction operands). Similarly, register names and memory addresses are not stored in the DDG, though they are shown in the figure for ease of reading.
- The same register/memory location can be mapped onto multiple nodes in the DDG, just as a given register or memory location can have multiple value instances during the execution. For example, in Figure 1, the value produced in register R1 is mapped onto nodes 6, 11, 16, 21, 26, one for each loop iteration.
- Each edge of the DDG is marked with the letter that represents the role of the operand in the instruction: *M* is a memory operand, *A* is an address operand, *P* is a regular operand, *B* is an operand used as a branch target, *F* is a function address operand, and *S* is a system call operand.

Function calls and returns are also represented in the DDG (not present in the example in Figure 1). Most of the semantics of function calls, such as setting up and tearing down the stack frame or parameter passing, that are already present in the assembly code are automatically included as part of the DDG. However, calling conventions cannot be extracted from the machine code and are explicitly specified in the DDG. For example, in the SPARC architecture, the register *R2* is used to store the return value of a function, and this must be incorporated in the DDG to analyze dependencies across function calls and returns. The DDG also incorporates dependencies caused by system calls (not present in the example in Figure 1). Our method for DDG construction is similar to the one proposed in [17].

4. Fault Model

This study considers the impact of faults in data values produced during the course of a program's execution. Our fault model assumes that any dynamic value in a program can be corrupted at the time of the value definition. This corresponds to an incorrect computation of the value mainly due to transient (or soft) errors and includes all values written to memory, registers, and the processor cache. Note that the assumed fault model also covers errors that arise due to some categories of software faults, e.g., *assignment/initialization* (an uninitialized or incorrectly initialized value is used) or *checking* (a check performed on the variable fails, which is equivalent to an incorrect value of a variable being used) [6].

Crash model. Since the ultimate goal is to ensure a crash-failure semantic for application processes, we first introduce a crash model. It is assumed that crashes can occur as a result of (i) illegal memory references (*SIGBUS* and *SIGSEGV*), (ii) divide-by-zero and overflow exceptions (*DIVBYZERO, OVERFLOW*), (iii) invocation of system calls with invalid arguments, or (iv) branch to an incorrect or illegal code (*SIGILL*). These four categories can be represented in the Dynamic Dependence Graph (DDG) described in the previous section as follows:

1. A value used as an address operand in a load or store instruction is corrupted and causes the reference to be misaligned or to be outside a valid memory region.
2. A value used in an arithmetic or logic operation is corrupted and causes a divide-by-zero exception or arithmetic overflow.
3. A value used as a system call operand is incorrect or the program does not have the permissions to perform a particular system call.
4. An operand used as the target of a branch or as the target address of an indirect function call is corrupted, causing the program to jump to an invalid region or to a valid (part of the application) but incorrect (from the point of view of the application semantic) region of code.

Usually, corruption of pointer data is much more likely to cause a crash than non-pointer data, as shown by earlier studies, e.g., [15]. Therefore, this study considers only crashes due to (i) corruption of values used as address operands of load/store instructions (the first category) or (ii) corruption of values used as targets of branches and function calls (the last category discussed above). While the model does not consider corruption of system call operands or operands of arithmetic and logic instructions, we found that in practice (i.e., in real programs) the percentage of crashes missed by the model is small.

Analysis of error propagation. The dynamic execution traces provided by the DDG are used to reason about error propagation from one value to another. It is assumed that a fault originating in a node (value) of the DDG can potentially propagate to all nodes that are successors of this node in the DDG. Unique error-propagation probabilities are not assigned to DDG edges (unlike in other studies, e.g., [9]); rather, it is assumed that errors can propagate along every arc with the probability of 1.

5. Metrics Derived from the Models

To strategically place detectors, we develop a set of metrics for selecting locations in the program that can provide high crash detection coverage. The metrics are derived based on the DDG of the program. To enable placement of detectors in the code, the notion of *static location of a value* is introduced. The static location of a value is defined as the address of the instruction that produces the value. The following metrics are employed:

- *Fanout*: The *fanout* of a node is the set of all immediate successors of the node in the DDG. In terms of values, it is the set of uses of the value represented by the node. The *fanout* of a node indicates how many nodes are directly impacted by an error in that node.
- *Lifetime:* The *lifetime* of a node is the maximum distance (in terms of dynamic instructions) between the node and its immediate successors. In terms of values, it is the maximum dynamic distance between the *def* and *use* of a value. The *lifetime* evaluates the reach of the error in the program's execution. This is because values with a long *lifetime* are typically global variables or global constants, and an error in these values can affect values that are distant from the current execution context of the program.
- *Execution:* The *execution* of a node is the number of times the static location (program counter) associated with the value is executed. *Execution* reflects the intuition that a location that is executed more frequently is a good place to embed a detector.
- *Propagation:* The *propagation* of a node is the number of nodes to which an error in this node propagates before causing a crash. The *propagation* metric is somewhat similar to the *fanout* metric, but while the *fanout* metric considers only the first level of error propagation, the *propagation* metric characterizes error propagation across multiple levels.
- *Cover:* The *cover* of a node is the number of nodes from which an error can propagate to a given node before causing a crash. Nodes with a high *cover* usually have many error-propagation paths passing through them, and consequently, these nodes are good locations for placing detectors to enable preemptive crash detection.

Since detectors are placed in the static code of the program, each node selected (based on the computed metrics) for a detector must be mapped onto the static locations in the program. Note that multiple nodes in the DDG can be mapped onto a single static location. Consequently, aggregation functions must be defined to compute overall metrics corresponding to a given static program location based on the metrics of the nodes that map onto this location. In the case of *fanout*, *propagation*, and *cover* metrics, the set-union operation is used to compute the aggregate set. The cardinality of the aggregate set is calculated as the aggregate *fanout*, *propagation*, and *cover* of that location, respectively. For lifetime and execution, the aggregate value of the metric at a location is computed as the average of the metric values of the nodes that map onto this location.

For the example in Figure 1, nodes 6, 11, 16, 21, and 26 map onto the value produced by the static instruction *ADDI R1, R1, 1*. The instruction has the following metric values:

- The *aggregate fanout* of the instruction is the cardinality of the union of the set of immediate successors of 6, 11, 16, 21 and 26, namely the cardinality of the set {5, 6,7, 10, 11, 12, 15, 16, 17, 20, 21, 22, 25, 27}, which is equal to 15.
- The *aggregate lifetime* of the instruction is the average of the lifetimes of the nodes 6, 11, 16, 21, and 26. The lifetime of each of these nodes is 4 dynamic instructions (the length of a

loop iteration), except for 26 for which it is only one dynamic instruction (the last loop iteration). Therefore, the aggregate lifetime of the instruction is 4.25.

- The *aggregate execution* value for the instruction is 5, as the loop is executed 5 times.

For computing the *propagation* and *cover* metrics, we need to locate the points at which the program can crash. The *crash-set* of a node in the DDG is the set of all nodes at which a crash can potentially occur due to an error in that node. The *crash-point* of a node is the earliest point in the error's propagation (not to be confused with the *propagation* metric) at which a crash can occur because of pointer corruption or corruption of a branch/function call target address (this follows from the crash model defined in Section 4, in which only corruptions of pointers and function/branch targets are assumed to cause crashes). For each node N in the DDG, we denote by *Crash(N)* the crash-point of N. In the rare case that a node has multiple crash points, we arbitrarily pick one of them to be Crash(N). If there is no crash due to a fault at N, we assume that *Crash(N)* is nil. For the example in Figure 1, the crash-points of nodes 6, 11, 16, 21, and 26 are nodes 5, 10, 15, 20, and 25 respectively, as these are used as address operands in the instruction *LW R3, A(R1)*. The *crash-distance* of a node (*CrashDist(N)*) is the distance between the node and its crash-point in the DDG, in terms of dynamic instructions.

Once the crash-distance is computed, the *propagation* set of a node/location N can be computed as the union of the *propagation* sets of the successor nodes of N, such that the distance from N to a node x in its *propagation* set (*dist(x,N)*) is less than or equal to the crash-distance of N (*CrashDist(N)*). The propagation set of a node also includes the node itself.

The *aggregate propagation* of a location can be computed as the cardinality of the union of the *propagation* sets of the nodes in the DDG that map onto this location. For the example in Figure 1, the *aggregate propagation* of the node corresponding to instruction *ADDI R1, R1, 1* is 10, as the union of the *propagation* sets of its DDG nodes 6, 11, 16, 21, and 26 is the set of nodes {6, 11, 16, 21, 26, 5, 10, 15, 20, 25}. Note that although nodes 7, 12, 1, 22, and 27 are successors of the nodes 6, 11, 16, 21, and 26, they do not appear in the *propagation* sets, as their distance from these nodes (4) is greater than the crash-distance of the nodes (2).

Once the *propagation* metric is computed, the *cover* metric can be computed as follows: *A node M is in the cover of N if and only if N belongs to the propagation of M.* This is because any fault in N must propagate to M before causing a crash if M belongs to the *cover* of N (by definition). In the example in Figure 1, the *aggregate cover* of the node corresponding to instruction *LW R3, R1(A)* is the cardinality of the union of the *cover* sets of its nodes in the DDG, namely 5, 10, 15, 20, and 25. This is the set {6, 11, 16, 21, 26}, as the nodes 5, 10, 15, 20, and 25 collectively appear in the *propagation* sets of nodes 6, 16, 11, 21, and 26. Hence, the *aggregate cover* is 5, which is the cardinality of the set.

6. Experimental Setup

This section describes the experimental infrastructure and application workload used to evaluate the model and the metrics. The experiment is divided into three parts:

1. *Tracing*: The application program is executed and a detailed execution trace is obtained containing all the dynamic dependencies, branches, and load/store instructions.
2. *Analysis:* The trace is analyzed, the dynamic dependence graph (DDG) is constructed, and the metrics for placing detectors are computed; this part is done offline.
3. *Fault Injection*: Fault injections are performed to evaluate the choice of the detector points. The values at the detector points are recorded and compared with the corresponding values in the *golden* (error-free) run of the application. Any deviation between the values in the golden run and the faulty run indicates successful detection of the error by the detection point.

6.1 Infrastructure

The tracing and fault injections are performed using a functional simulator in the *SimpleScalar* family of processor simulators [4]. The simulator allows fine-grained tracing of the application without perturbing its state or modifying application code. It 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 to the trace. The trace is analyzed offline by specialized scripts to construct the DDG and compute the metrics for placing detectors in the code. The top 100 points according to each metric are chosen as locations for inserting detectors.

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. When the tag value of the current instruction equals the value of a specified fault location, a fault is injected by flipping a single bit in the value produced by the current instruction. Once a fault is injected, the execution sequence is monitored to see whether a detector location is reached. If so, the value at the detector location is written to a file for offline comparison with the golden run of the application. Table 2 shows the errors detected by the simulator and their corresponding consequence in a real system. It also explains the detection mechanism in the simulator and in the real system.

6.2 Application Programs

The system is evaluated with four programs from the Siemens suite [12] and two programs from the SPEC95 benchmark suite. These benchmark applications range from a few hundred lines of code (Siemens) to tens of thousands of lines of code (SPEC95). A brief description of benchmarks is given in Table 3. tcas from the Siemens suite is omitted, as it is very small (<200 lines of C code), and separation among the different metrics used in the study was insufficient.

Each of these applications is executed for three inputs. For the Siemens programs, the inputs are chosen from the provided set of inputs. For *gcc95* and *perl*, we create inputs of reduced size (compared to the original SPEC workloads), since our analysis scripts were unable to handle the extremely large dynamic traces of the SPEC workloads. Also, for the SPEC benchmarks, infrequently executed dynamic control paths that contributed to less than 20% of the cumulative execution time are removed from the DDG (this constitutes approximately 80% of the program paths).

Table 2: Types of errors detected by simulator and their real-world consequences

Type of error detected	Consequence in a real system	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 DIV 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 continuously issues instructions and never terminates	Program executes double the number of instructions compared with the golden run
Indefinite wait due to blocking system calls or I/O	Program hang (deadlock); program stops issuing instructions and never terminates	Program execution takes substantially longer (five times) than the golden run
Incorrect output	Fail-silent violation (silent data corruption)	Compare outputs at the end of the run

Table 3: Benchmarks and their descriptions

Benchmark	Suite	Description
Replace	Siemens	Searches a text file for a regular expression and replaces all occurrences of the expression with a specified string
Schedule2	Siemens	A priority scheduler for multiple job tasks
Print_tokens	Siemens	Breaks the input stream into a series of lexical tokens according to pre-specified rules
Tot_info	Siemens	Offers a series of data analysis functions
Gcc95	SPEC95	The *gcc* compiler
Perl	SPEC95	The *perl* interpreter

For each program, the dynamic trace from one of the inputs is chosen to build the DDG and to perform the analysis to choose detector points (the top 100 locations according to each metric). Fault injections are then performed at randomly chosen values in the application's execution for all three inputs. For each application, input, and metric used to choose detector points, faults are injected at 500 random locations, randomly flipping a single bit of a value. This is done 10 times for each location, for a total of 5000 fault injections for each combination of application, input, and metric. One fault is injected per run to eliminate the possibility of latent errors due to faults injected earlier.

7. Results

The results obtained from the experiments are analyzed with the objective of answering the following questions:

- What is the detection coverage provided by individual detectors placed according to a given metric?
- What is the rate of false-positives of individual detectors placed according to a given metric?
- What is the detection coverage provided jointly by multiple detectors placed according to a given metric?
- What is the rate of false-positives of multiple detectors placed according to a given metric?

7.1 Detection Capability of Metrics for Single Detectors

This section evaluates the detection coverage provided by individual detectors placed according to different metrics. All results represent the average calculated for each application across three inputs. The detector points that registered a value deviation for an injection are associated with the outcome of the injection. The results for each outcome category (crash, hang, fail-silent violation, success) are normalized across the total number of errors observed under that category (for each benchmark-metric combination). The results for crashes, successes (false-positives) and fail-silent violations are shown in Figures 2, 3, and 4, respectively. The results for hangs are not shown due to space limitations. The interested reader may refer to [22] for the same.

The following can be concluded from the graphs:

- Detectors placed according to the *fanout* and *propagation* metrics are the best at detecting crashes. They are followed by detectors placed according to the *cover* metric (see Figure 2). The coverage provided by *fanout* and *propagation* detectors is more than 90% for the Siemens benchmarks (except for *tot_info*). For the SPEC benchmarks, the coverage is around 50-60%.
- The percentage of false-positives is small, less than 2% for all benchmarks except *replace* (see Figure 3). The higher false-positive rates for *gcc95* and *perl* are registered by detectors placed using *fanout* (1.5%) and *propagation* (2%) metrics.
- Although the detector points were chosen to support crash detection, they also detect a significant percentage of fail-silent violations (30-70% for detectors placed using *fanout* and *propagation* metrics, as shown in Figure 4) and hangs (60-90%).

7.1.1 Discussion

Locations having high *fanout* and *propagation* are responsible for propagating errors to a large number of places in the DDG, and it is likely that at least one of the propagated errors causes a crash. Detectors placed using *fanout* do marginally better than those inserted using *propagation*. There are two key reasons for the differences: (i) *Propagation* relies on the accuracy of the crash model in deciding on the further propagation of the error, while *fanout* does not take the crash model into account and is more conservative. (ii) Locations with a high *fanout* are often stack or frame pointers. The program accesses these locations frequently, hence an error is likely to cause a crash.

The SPEC benchmarks execute more than a million dynamic instructions, while the Siemens benchmarks typically execute less than 100,000 (only *tot_info* in the Siemens suite executes between 100,000 and a million instructions). As a result, the probability of the error reaching the detector is higher in the case of the Siemens benchmarks than for the SPEC95 benchmarks. Hence, the detection coverage for *replace*, *schedule2,* and *print_tokens* ranges between 80% and 90%, compared with 50-70% for *gcc*, *perl*, and *tot_info*.

The *execution* metric is a good indicator for placing detectors in the Siemens benchmarks, where infrequently executed paths are not pruned. However, it does not perform well in the SPEC benchmarks, where paths that contribute to less than 80% of the execution time are already removed.

Detectors placed using the *lifetime* metric do not have high crash-detection coverage, as the error is likely to remain latent for a long time in a high-lifetime node and is unlikely to cause a crash. The lower effectiveness of detectors placed using the *cover* metric compared to *propagation* and *fanout* stems from the fact that *cover* aims at placing detectors along paths leading to potential crash-points, while *propagation* and *fanout* place detectors along paths that can potentially spawn errors in many nodes. Typically, the number of locations with high *fanout* or *propagation* is small (these metrics follow a *Pareto-Zipf* law like distribution), while the number of potential crash-points of the application is much larger. This result shows that *it is more beneficial to place detectors to protect these few, highly sensitive values, than it is to place detectors along paths that lead to potential crash points.*

The false-positive rate for the metrics is less than 2% for all benchmarks except *replace*. A false-positive means that the error was detected by a detector point, but the program completed successfully (and produced correct output). The number of instructions executed by *replace* is around 10,000, and hence the probability of an error reaching the detector point is high even if the error does not trigger a failure.

7.2 Detection Capability of Metrics for Multiple Detectors

The previous section considered the detection provided by placing a single detector in each of the benchmark programs. For the Siemens benchmarks (except *tot_info*), this was sufficient to provide coverage of 90%. However, for applications such as *gcc95* and *perl*, a single detector could achieve only up to 60% coverage. In this section, we evaluate the coverage provided jointly by multiple detectors placed in *gcc95* and *perl*.

The top 100 detector locations selected by each metric are grouped into *bins* of a predefined size, and the cumulative coverage of detectors placed at locations indicated by a bin is evaluated. For example, to evaluate the coverage of the *fanout* metric with a bin size of 10, the top 100 locations with the highest *fanout* are arranged in decreasing order by their *fanout* value. The top 10 locations are then grouped into *bin 1*, the next 10 locations into *bin 2,* and so on up to *bin 10*. The crash-detection coverage of each bin as a whole is evaluated. The average coverage of the 10 bins is the crash-detection coverage for the *fanout* metric with a bin size of 10.

The results for crash detection, false-positives, and fail-silent violations are shown in Figures 5 through 10 as a function of the bin size for *gcc95* and *perl*.

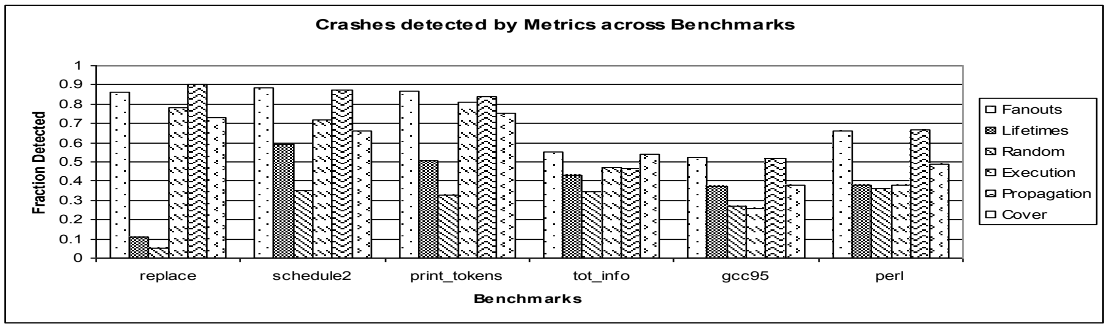

Figure 2: Crashes detected by detectors across benchmarks

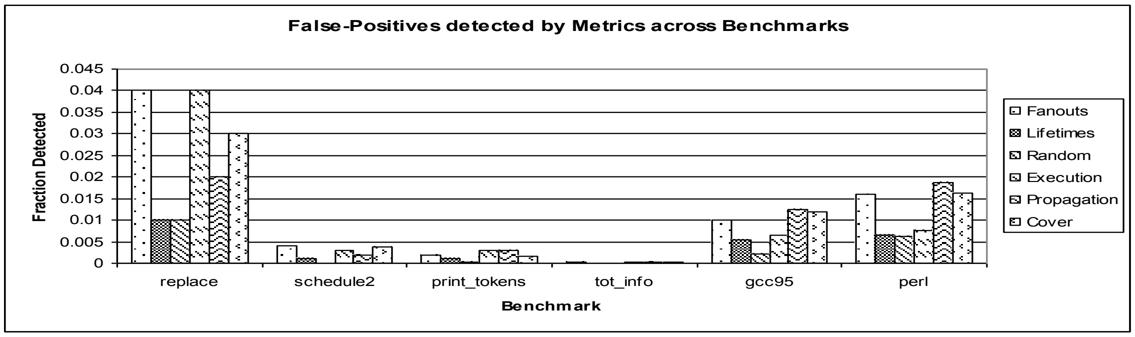

Figure 3: False-Positives detected by detectors across benchmarks

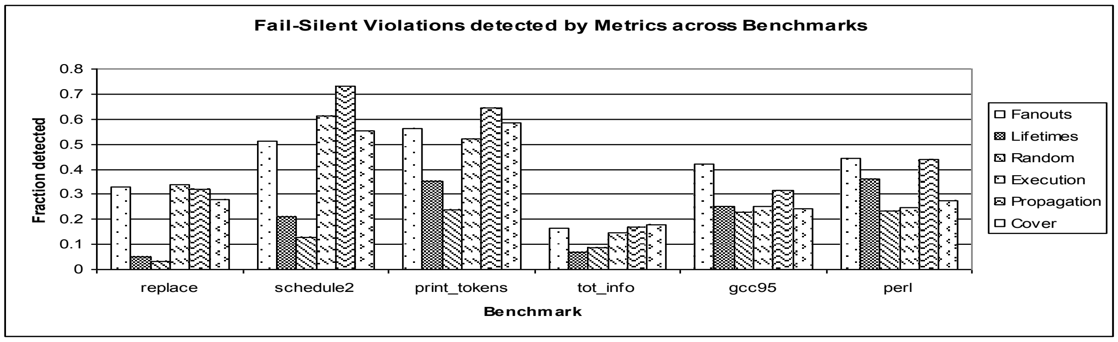

Figure 4: Fail-silent violations detected by detectors across benchmarks

The results for *gcc95* are summarized below, and similar trends are observed for *perl.*

- For detectors placed using *fanout* and *propagation*, the crash-detection coverage is less than 60% when the bin size is 1 (as discussed in section 7.1). Increasing the bin size to 10 improves coverage to 80% (see Figure 5).
- For a bin size of 1, the coverage provided by detectors placed according to *lifetime* is less than 40% (as discussed in Section 7.1). However, for a bin size of 10, the coverage is almost equal to that provided by detectors placed according to *fanout* and *propagation* metrics. For a bin size of 25 and 100, it even surpasses the coverage of detectors placed using *fanout,* providing coverage values of 90% and 99%, respectively (see Figure 5).
- The percentage of false-positives also increases with increasing bin-size, but not as much as the crash-detection coverage. For example, for detectors placed using the *fanout* metric, the coverage is around 80% when the bin size is 10, but the number of false-positives remains around 5% (see Figure 7).
- The increase in the false-positive rate for *lifetime* is much less than it is for *fanout*. The false-positive percentage for *lifetime* is only 5% for a bin size of 100 compared to 10% for *fanout* for the same bin size. When 10 or more detectors are considered, placement based on the *lifetime* metric provides the best coverage and the lowest rate of false-positives (see Figure 7).
- Random detector placement provides coverage of 95% (see Figure 5) when the bin size is 100. Further, it has the smallest percentage of false-positives (2.5%, see Figure 7), making random placement of multiple detectors a good choice when minimizing false-positives is critical.
- The fail-silent violation coverage is highest for detectors placed using the *fanout* metric (70% for a bin size of 10, see Figure 9). For a bin size of 100, detectors placed using the *execution* metric surpass the detectors placed using *fanout.*

7.2.1 Discussion

For all the metrics, the coverage increases with increases in the bin size as the number of detector points increases. This increase in the coverage as bin size increases flattens out, however, since there is considerable overlap among the multiple detector points in detecting crashes. For example, for detectors placed using the *fanout* metric, grouping detectors into bins of size 5 increases the coverage to 75% (from the 60% coverage provided by individual detectors). However, the increase in coverage is less when the bin size increases to 10 (coverage 80%).

Detectors at locations with a high *lifetime* provide limited coverage individually, but several of them jointly achieve very high coverage. This is because each detection point covers a different set of errors. Closer analysis of the results indicates that there is usually one *hot-detector* in each bin. The hot-detector detects the majority of errors covered by that bin, and the other detectors complement the coverage by detecting errors that escape the hot-detector. These errors are also not easily detectable by the detectors placed using other metrics.

When detectors are grouped into bins, the rate of the false-positives increases, as there are more detector-points that are likely to capture the error even if no crash occurs. However, for detectors chosen using the *lifetime* metric, each detector point by itself has a lower detection rate than detectors selected using other metrics, and hence the number of false-positives is also low. When combined together in bins, the coverage numbers are additive, whereas the false-positives are not.

The coverage provided by random detector placement is the lowest of all metrics (with the exception of *execution*) for bin sizes up to 25 (for *gcc*). For bin-size beyond 25, the coverage provided by the randomly placed detectors increases sharply and is comparable to the coverage provided by the detectors placed using other metrics (for a bin size of 100). In the top 100 detector points picked by the random metric, there are a few points that provide very high coverage. Consequently, if the bins are large enough, it is likely that at least one such high coverage point gets included in each bin.

For *gcc95*, 100 detectors correspond to about 1% of the locations on its hot-paths, while for *perl*, 25 detectors correspond to 1% of its hot-paths locations. Hence, by placing detectors at less than 1% of the hot-paths in both applications, it is possible to obtain up to 99% coverage.

7.3 Summary of Results

This section summarizes the results from sections 7.1 and 7.2 as follows:

- Detectors placed using the *fanout* metric have the best coverage in the program when single detectors are considered. The coverage provided is 90% for the Siemens benchmarks and 50-60 % for the SPEC benchmarks.
- When multiple detectors are placed using the *fanout* metric, the coverage increases to 97% by inserting detectors at less than 1% of the hot-paths. In the multiple detector case, the coverage provided by the detectors placed using the *lifetime* metric is higher than the coverage provided by detectors placed using the *fanout* metric (for 10 or more detectors).
- Random assertion placement on the hot-paths of the application can provide coverage of up to 95% with a low rate of false-positives if detectors are placed at 1% of the hot-path locations.

8. Conclusions and Future Work

This paper explores the problem of detector placement in programs to preemptively detect crashes due to errors in data values used in the program. A model for error propagation and crashes is developed, and metrics for placing detectors are derived from the model. The metrics are evaluated on six applications, including two SPEC95 benchmarks. It is found that strategic placement of detectors can increase crash coverage by an order-of-magnitude compared to random placement, with a low percentage of false-positives. Though the placement of detectors was geared toward detecting crashes, the detectors also detected a sizeable percentage of fail-silent violations and hangs.

Future work will involve integrating the analysis into a compiler and automatically deriving the actual detectors for checking. We also plan to integrate the detectors as hardware modules in the Reliability and Security Engine (RSE) [21].

Acknowledgments

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.

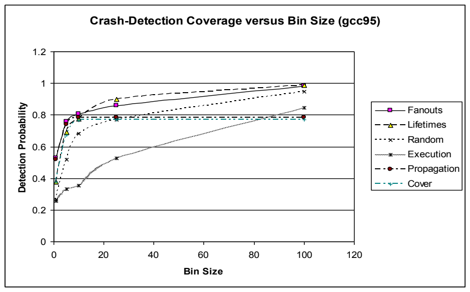

Figure 5: Effect of bin size on crash detection coverage for *gcc* for different metrics

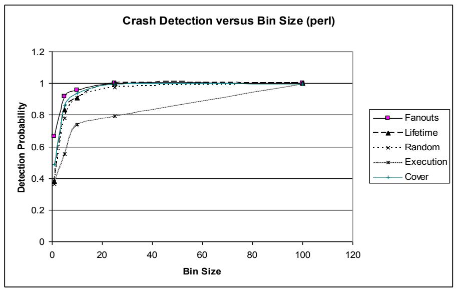

Figure 6: Effect of bin size on crash detection coverage for *perl* for different metrics

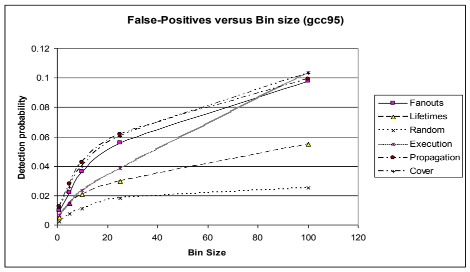

Figure 7: Effect of bin size on false-positive rate for *gcc* for different metrics

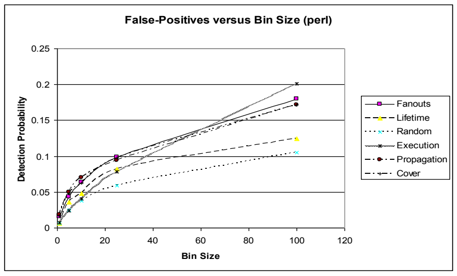

Figure 8: Effect of bin size on false positive rate for *perl* for different metrics

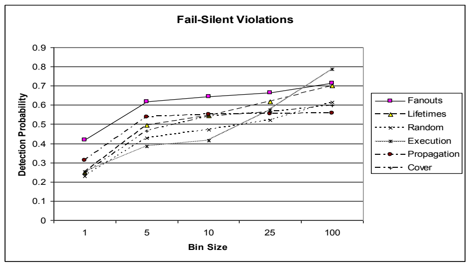

Figure 9: Effect of bin size on fail-silent violation coverage for *gcc* for different metrics

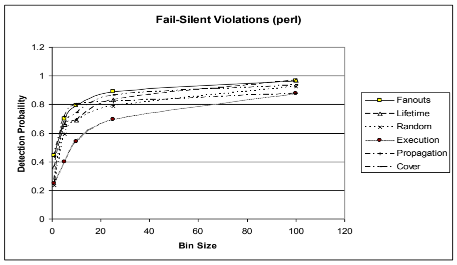

Figure 10: Effect of bin size on fail-silent violation coverage for *perl* for different metrics

References

[1] Aho, R. Sethi, and J. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, Reading, MA, 1986.

[2] C. Basile, L. Wang, Z. Kalbarczyk, and R. K. Iyer, Group Communication Protocols under Errors, Symp. on Reliable Distributed Systems (SRDS) 2003.

[3] A. Benso, S. Carlo, G. Natale, L. Tagliaferri, and P. Prinetto, Validation of a Software Dependability Tool via Fault Injection Experiments, 7th Intl. On-Line Testing Workshop, 2001.

[4] 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.

[5] S. Chandra and P.M Chen, How Fail-Stop Are Faulty Programs?, Proc. 28th Intl. Symposium on Fault-Tolerant Computing (FTCS-28), 1998.

[6] R. Chillarege, W-L Kao, and R. Condit, Defect Type and its Impact on the Growth Curve, Proc. 13th Intl. Conference on Software Engineering, 1991.

[7] Compaq Computer Corp, Data Integrity for Compaq NonStop Himalaya Servers, http://nonstop.compaq.com, 1999.

[8] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, Dynamically Discovering Likely Program Invariants to Support Program Evolution, IEEE Trans. on Software Engineering, 27(2), 2001.

[9] T. Goradia, Dynamic Impact Analysis: A Cost-Effective Technique to Enforce Error-Propagation, ISSTA 1993.

[10] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, Characterization of Linux Kernel Behavior under Errors, Proc. Intl. Conference on Dependable Systems and Networks (DSN), 2003.

[11] M. Hiller, Executable Detectors for Detecting Data Errors in Embedded Control Systems, Proc. Intl. Conference on Dependable Systems and Networks (DSN), 2000.

[12] M. Hiller, A. Jhumka, and N. Suri, On the Placement of Software Mechanisms for Detection of Data Errors, Proc. Intl. Conference on Dependable Systems and Networks (DSN), 2002.

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

[14] R.K Iyer, D.J Rosetti, and M.C. Hseuh, Measurement and Modeling of Computer Reliability as Affected by System Activity, ACM Trans. on Computer Systems, 4(3), 1986.

[15] W. Kao, R. K. Iyer, and D. Tang, FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults, IEEE Trans. on Soft-ware Engineering, 19(11), 1993.

[16] S. Narayanan, S. Son, M. Kandemir, and F. Li, Using Loop Invariants to Fight Soft Errors in Data Caches, Proc. Asia and South Pacific Design Automation Conference (ASP-DAC'05), 2005.

[17] N. Nethercote and A. Mycroft, Redux: A Dynamic Dataflow Tracer, Proc. of 3rd Workshop on Runtime Verification (RV'03), 2003.

[18] T. Slegel, et al., IBM's S/390 G5 Microprocessor Design, IEEE Micro, 19(2), 1999.

[19] A. Thakur, Measurement and Analysis of Failures in Computer Systems, M.S Thesis, University of Illinois, UILU-ENG-97, September 1997.

[20] J. Voas and K. Miller, The Avalanche Paradigm: An Experimental Software Programming Technique for Improving Fault Tolerance, Proc. of ECBS, 1999.

[21] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu, An Architectural Framework for Providing Reliability and Security Support, Proc. Intl. Conference on Dependable Systems and Networks (DSN), 2004.

[22] K.Pattabiraman, Z.Kalbarczyk and R.K. Iyer, Application-Based Metrics for Strategic Placement of Detectors, Technical Report, University of Illinois at Urbana-Champaign, 2005.