Automated Derivation of Application-aware Error Detectors Using Static Analysis

Karthik Pattabiraman, Zbigniew Kalbarczyk and Ravishankar K. Iyer
Center for Reliable and High Performance Computing, University of Illinois (Urbana-Champaign)
{pattabir, kalbar, rkiyer} @uiuc.edu

Abstract

This paper presents a technique to derive and implement error detectors to protect an application from data errors. The error detectors are derived automatically using compiler-based static analysis from the backward program slice of critical variables in the program. Critical variables are defined as those that are highly sensitive to errors, and deriving error detectors for these variables provides high coverage for errors in any data value used in the program. The error detectors take the form of checking expressions and are optimized for each control flow path followed at runtime. The derived detectors are implemented using a combination of hardware and software. Experiments show that the derived detectors incur low performance overheads while achieving high error-detection coverage.

1. Introduction

This paper presents a methodology to derive error detectors for an application based on compiler (static) analysis. The derived detectors protect the application from data errors. A data error is defined as a divergence in the data values used in the application from an error-free run of the program. Data errors can result from incorrect computation and would not be caught by generic techniques such as ECC in memory. They can also arise due to software defects (bugs).

In the past, static analysis [1] and dynamic analysis [2] approaches have been proposed to find bugs in programs. These approaches have proven effective in finding known kinds of errors prior to deployment of the application in an operational environment. However, studies have shown that the kinds of errors encountered by applications in operational settings are often subtle errors (such as timing and synchronization errors) [8], which are not caught by static and dynamic methods. Furthermore, programs upon encountering an error, may execute for billions of cycles before crashing (if they crash), during which time the error may propagate to permanent state [15]. In order to detect runtime errors, we need mechanisms that can provide high-coverage, low-latency (rapid) error detection to preempt uncontrolled system crash/hang and prevent error propagation that can lead to state corruption.

Duplication has traditionally been used to provide high-coverage at runtime for software errors and hardware-errors. However, in order to prevent error-propagation and preempt crashes, a comparison needs to be performed after every instruction, which in turn results in high performance overhead. Therefore, duplication approaches compare the results of replicated instructions at selected program points such as stores to memory [4] [14]. While this reduces the performance overhead of duplication, it sacrifices coverage as the program may crash before reaching the comparison point. Further, duplication-based techniques detect all errors that manifest in instructions and data. It has been found that less than 50% of these errors result in application failure (crash, hang or incorrect output) [11]. Therefore, more than 50% of the errors detected by duplication are wasteful from the application's perspective.

The main contribution of this paper is an approach to derive runtime error detectors based on application properties extracted using static analysis. The derived checks preempt crashes and provide high-coverage in detecting errors that result in application failures.

The coverage of the derived detectors is evaluated using fault-injection experiments. The key findings are:

- Derived detectors detect around 75% of errors that propagate and cause crashes. The percentage of benign errors detected is less than 3%.
- The average performance overhead of the derived detectors across 14 benchmark applications is 33%.

2. Fault Model

Hardware transient errors that results in corruption of architectural state are considered. Examples of such errors are:

- **Errors in Instruction Fetch and Decode:** Either the wrong instruction is fetched, (OR) a correct instruction is decoded incorrectly resulting in data value corruption.

- **Errors in Execute and Memory Units:** An ALU instruction is executed incorrectly inside a functional unit, (OR) the wrong memory address is computed for a load/store instruction, resulting in value corruption.
- **Errors in Cache/Memory/Register File Errors:** A value in the cache, memory, or register file experiences a soft error that causes it to be incorrectly interpreted in the program (assuming that ECC is not used).

Software transient errors such as **buffer overflows** (memory errors) and **race conditions** (timing errors), which can corrupt data values used in the program, are also considered.

3. Approach

This section presents an overview of the detector derivation approach. The approach is based on the technique of program slicing.

3.1 Terms and Definitions

Backward Program Slice of a variable at a program location is defined as the set of all program statements/instructions that can affect the value of the variable at that program location [6].

Critical variable: A program variable that exhibits high sensitivity to random data errors in the application is a critical variable. Placing checks on critical variables achieves high detection coverage.

Checking expression: *A checking expression is a sequence of instructions that recomputes the critical variable, and is optimized aggressively and differently from the rest of the program code.* The instruction sequence is computed from the backward slice of the critical variable for a specific control path in the program.

3.2 Steps in Detector Derivation

The main steps in the derivation of error detectors are as follows:

3.2.1 Identification of critical variables. The critical variables are identified based on an analysis of the dynamic dependence graph of the program presented in [3]. This analysis is carried out on a per-function basis in the program i.e. each function in the program is considered separately for identification of critical variables.

3.2.2 Computation of backward slice of critical variables. A backward traversal of the static dependence graph of the program is performed starting from the instruction that computes the value of the critical variable going back to the beginning of the function. The slice is specialized for each acyclic control path that reaches the computation of the critical variable from the top of the function. The slicing algorithm used is a static slicing technique that considers all possible dependences between instructions in the program regardless of program inputs.

3.2.3 Check derivation, insertion, instrumentation.

- *Check derivation***:** The specialized backward slice for each control path is optimized considering only the instructions on the corresponding path, to form the checking expression.
- *Check insertion:* The checking expression is inserted in the program immediately after the computation of the critical variable (*check placement point).*
- *Instrumentation:* Program is instrumented to track control-paths followed at runtime so as to choose the checking expression for that specific control path.

3.2.4. Runtime checking in hardware and software. The control path followed is tracked by the inserted instrumentation in hardware at runtime. The path-specific inserted checks are executed at appropriate points in the execution depending on the runtime control path. The checks recompute the value of the critical variable for the runtime control path. The recomputed value is compared with the original value computed by the main program. In case of a mismatch, the original program is stopped and recovery is initiated.

There are two sources of runtime overhead for the detector:

(1) Path Tracking: The overhead of tracking paths is significant (4x) when done in software. Therefore, a prototype implementation of path tracking is performed in hardware. This hardware is integrated with the Reliability and Security Engine (RSE) [12]. RSE is a hardware framework that provides a plug-and-play environment for including modules that can perform a variety of checking and monitoring tasks in the processor's data-path. The path-tracking hardware is implemented as a module in the RSE. Due to space constraints, the design of the path-tracking module is not presented in this paper but may be found in [13].

(2) Checking: In order to further reduce the performance overhead, the check execution itself can be moved to hardware. This would involve implementing expressions directly in the RSE and is a direction for future work.

4. Detector Derivation

The derivation of detectors is done by introducing a new pass into the LLVM compiler [7], called the *Value Recomputation Pass (VRP).* The VRP performs the backward slicing starting from the instruction that

computes the value of the critical variable to the beginning of the function. It also performs check derivation, insertion and instrumentation. The output of the pass is provided as input to other optimization passes in LLVM. *By extracting the path-specific backward slice and exposing it to other optimization passes in the compiler, the Value Recomputation pass enables aggressive compiler optimizations to be performed on the slice that would not be possible otherwise.*

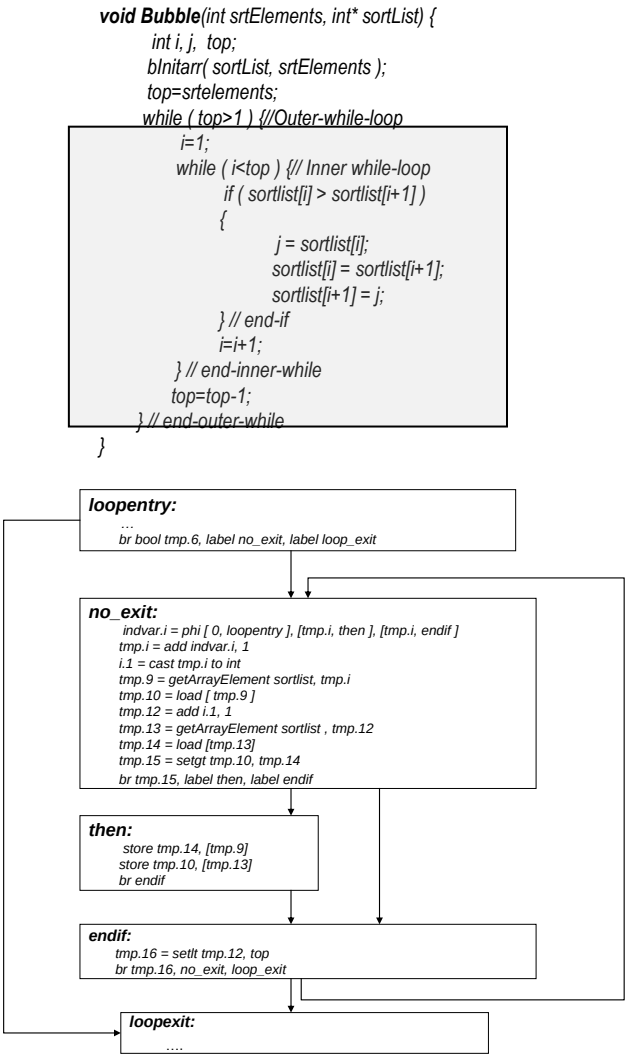

Figure 1: Bubble sort code fragment and intermediate code corresponding to inner loop

Figure 1 shows the source and LLVM intermediate code (SSA form [9]) for the inner while loop of a bubble sort program. In SSA form, each variable (value) is defined exactly once in the program, and the definition is assigned a unique name [9], which makes it easy to identify dependences among instructions.

In Figure 1, assume that the variable *tmp.10* has been identified as a critical variable. The final outcome after running the VRP and optimization passes is shown in Figure 2 for the computation of the critical variable *tmp.10*. The backward slice of this variable consists of the instructions that compute the values of *tmp.9, tmp.i, indvar.i.* These instructions are specialized and optimized depending on the control path executed (*path0 and path1*).

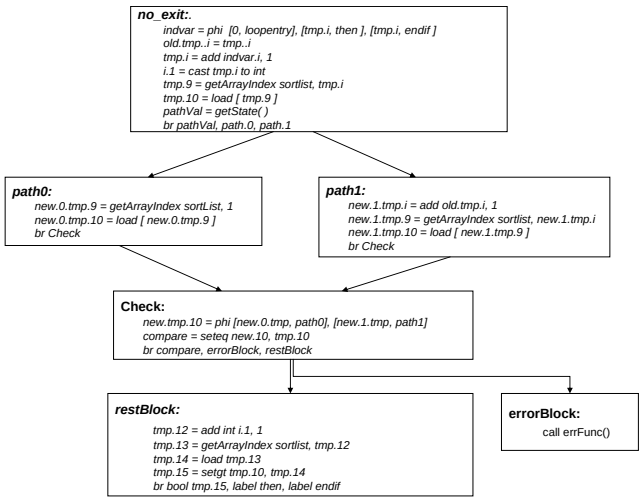

Figure 2: Transformations introduced by the VRP passes for the Bubble sort code fragment

4.1 Algorithm

The instruction that computes the critical variable is called the critical instruction. In order to derive the backward program slice, a backward traversal of the Static Dependence Graph (SDG) is performed starting from the critical instruction. The traversal continues until one of the following conditions is met, (1) The beginning of the current function is reached (only intra-procedural slices are considered) or (2) A basic block that had been previously encountered in the backward traversal is revisited (loops are not recomputed) or (3) The critical instruction occurs in-between the producer instruction of the dependence and the consumer instruction of the dependence (only previous loop iterations are considered when traversing loop-carried dependences) or (4) A memory dependence is encountered. The rationale for each of these cases is presented below**:**

- *Intra-procedural Slices:* It is sufficient to consider intra-procedural slices in the backward traversal because each function is considered separately

for the detector placement analysis. For example in Figure 1, the array *sortList* is passed in as an argument to the function from the *main* function. The slice does not include the computation of *sortList* in main. If *sortList* is a critical variable in the *main* function, then a check will be placed for the variable in the *main* function.

- *No recomputation of loops:* During the backward traversal, if a dependence within a loop is encountered, the loop is not recomputed in the checking expression. Instead, the check is broken into two checks, one placed on the critical variable and one on the variable that affects the critical variable within the loop.
- *Only the previous loop iteration is considered in traversing loop carried dependences:* When a loop-carried-dependence across two or more iterations is encountered, the dependence is truncated and the loop dependence is not included in the slice. This is because duplicating across multiple loop iterations can involve loop unrolling or buffering intermediate values that are rewritten in the loop. Instead, the check is broken into two checks, one for the dependence-generating variable and one for the critical variable.
- *Memory Dependences not considered.* While LLVM does not represent memory objects in SSA form, it promotes most memory objects to registers prior to running a pass (including the *Value Recomputation pass).* Since there is an unbounded number of virtual registers for storing variables in SSA form, the compiler is not constrained by the number of physical registers.

The details of the Value Recomputation Pass are not presented due to space constraints and may be found in the technical report version of this paper [13].

4.2 Derived Checks

The VRP creates two different instruction sequences to compute the value of the critical variable corresponding to the control paths in the code. The first control path corresponds to the control transfer from the basic block *loopentry* to the basic block *no_exit* in Figure 1. The optimized set of instructions corresponding to the first control path is encoded as a checking expression in the block *path0* in Figure 2.

The second control path corresponds to the control transfer from the basic block *endif* to the basic block *no_exit* in Figure 1. The optimized set of instructions corresponding to the first control path is encoded as a checking expression in the block *path1* in Figure 2.

The instructions in the basic blocks path0 and path1 recompute the value of the critical variable tmp.10. These instruction sequences constitute the checking expressions for the critical variable tmp.10. The basic block *Check* in Figure 2(c) compares the value computed by the checking expressions to the value computed in the original program. A mismatch signals an error and the appropriate error handler is invoked in the basic block *error*. Otherwise, control is transferred to the basic block *restBlock*, which contains the instructions following the computation of *tmp.10*.

5. Experimental Setup

This section describes the mechanisms for measurement of performance and coverage provided by the proposed technique.

5.1 Performance Measurements

The technique is evaluated with 9 programs from the Stanford benchmark suite and 5 programs from the Olden benchmark suite [10] . All experiments are carried out on a single processor P4 machine with 1GB RAM and 2.0Ghz clock running on the Linux operating system. Since the path-tracking is done in hardware, the overhead of tracking paths is negligible. The main sources of performance overhead for the technique are:

- *Modification overhead:* Performance overhead due to the extra code introduced by the VRP pass.
- *Checking overhead*: Performance overhead of executing the instructions in each check to recompute the critical variable.

5.2 Coverage Measurements

Fault Injection. Faults are injected into the application code to measure the coverage of the proposed technique. The fault-injection methodology inserts calls to a special *faultInject* function (at compile-time) after the computation of each program variable in the original program, with the value of the variable passed as an argument to the *faultInject* function. At runtime, the call to the *faultInject* function corrupts the value of a single program variable by flipping a single bit in its value. The value into which the fault is injected is chosen at random from the entire set of dynamic values used in an error-free execution of the program. In order to ensure controllability, only a single fault is injected in each execution of the application.

Error Detection. After a fault is injected, the following program outcomes are possible: (1) abnormal program termination (crash), (2) program continues and produces correct output (success), (3) program continues and produces incorrect output (fail-silent violation) or (4) program timeouts (hang).

The injected fault may also cause one of the inserted detectors to detect the error and flag a violation. When a violation is flagged, the program is allowed to continue (although in reality it would be stopped) so

that the final outcome of the program can be observed. The coverage of the detector is classified based on the observed outcome. For example, a detector is said to detect a crash if the detector upon encountering the error, flags a violation, and the program crashes.

Error Propagation. Our goal is to measure the effectiveness of the detectors in detecting errors that propagate before causing the program to crash. For errors that do not propagate before the crash, the crash itself may be considered the detection mechanism. Hence, coverage provided by the derived detectors for non-propagated errors is not reported.

In the experiments, error propagation is tracked by observing whether an instruction that uses the erroneous variable's value (according to the static data dependence graph of the program) is executed after the fault has been injected. If the original value into which the error was injected is overwritten, the error propagation is no longer tracked.

6. Results

This section presents the performance and coverage results obtained from the experimental evaluation of the proposed technique. The results are reported for the case when 5 critical variables were chosen in each function by the detector placement analysis.

6.1 Performance

The performance results are shown in Figure 3. The main results are summarized below:

- The average checking overhead introduced by the detectors is 25%, while the average code modification overhead is 8%. The total performance overhead is therefore 33%.
- The worst-case overheads incurred are in the case of *tsp*, which has a total overhead of nearly 80%. This is because *tsp* is a compute-intensive program involving tight loops. Placing checks within a loop introduces extra branches, and therefore increases its overhead.

6.2 Coverage

The coverage results (reported in percentages) are reported in Table 1. For each application, 1000 faults are injected, one in each execution of the application. A blank entry in the table indicates that no faults of the type were manifested for the application. For example, no hangs were manifested for *IntMM* in our experiment. Only program crashes that exhibit error propagation are considered. The numbers within the braces in this column indicate the percentage of propagated, crash-causing errors that are detected before propagation. The results in Table 1 show that:

- The derived detectors detect 77% of errors that propagate and crash the program. 64% of crash-causing errors that propagate are detected before first propagation. These correspond to 83% of the propagated crash-causing errors that are detected.
- The derived detectors detect 41% of errors that result in fail-silent violations (incorrect outputs) and 35% of errors that result in hangs.
- The number of benign errors detected is 2.5% on average. These errors have no effect on the execution of the application.
- The worst-case coverage for crashes (that exhibit error propagation) is obtained in the case of the Olden program *health* (39%). The *health* program is allocation-intensive, and spends a substantial fraction (over 50%) of its time in *malloc* calls. Our technique does not protect the return value of *mallocs* as duplicating *malloc* calls may change the semantics of the program. Further, the technique does not place detectors within the body of the *malloc* function, as it does not have access to the source-code of library functions. This is not an inherent limitation of our technique, and can be overcome by placing detectors inside library functions (by the library developer).

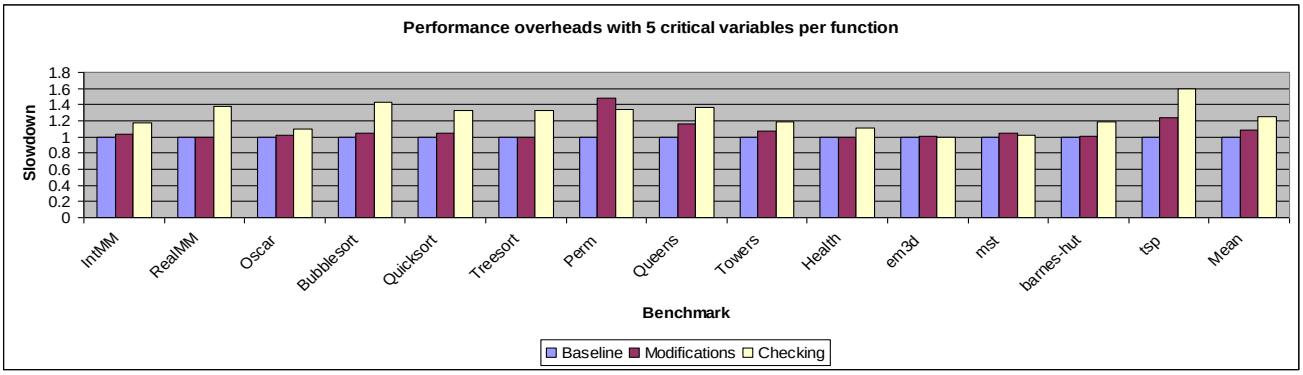

Figure 3: Performance overhead of executing checks in software (5 critical variables per function)

Table 1: Error-detection coverage of detectors

Apps	Propagated Crashes (%)	FSV (%)	Hang (%)	Success (%)
IntMM	100 *(97)*	100		9
RealMM	100 *(98)*			0
FFT	57 *(34)*	7	60	0.5
Quicksort	90 (57)	44	100	4
Bubblesort	100 *(73)*	100	0	5
Treesort	75 *(68)*	50		3
Perm	100 *(55)*	16		0.9
Queens	79 *(61)*	20		3
Towers	79 *(78)*	39	100	2
Health	39 *(39)*	0	0	0
Em3d	79 *(79)*			1
Mst	83 *(53)*	79	0	5
Barnes-Hut	49 *(39)*		23	
Tsp	64 (64)		0	0
Average	**77 (64)**	**41**	**35**	**2.5**

6.3 Discussion

The results indicate that our technique can achieve 75-80% coverage for errors that propagate and cause the program to crash. Full-duplication approaches can provide 100% coverage if they perform comparisons after each instruction. In practice, this is very expensive and full-duplication approaches compare instructions before store and branch instructions [4][14]. In this optimized mode of execution, the coverage provided by full-duplication is less than 100%. Studies that describe these techniques do not quantify the detection coverage in terms of error propagation, so a direct comparison with our technique is not possible. Further, the performance overhead of the technique is only 33 %, compared to full-duplication in software, which incurs an overhead of 60-100% [4][14].

An important aspect of the technique is that it detects just 2.5 % of benign errors in an application. In contrast, full duplication techniques detect between 50-60% of benign errors in the program [11].

7. Conclusions

This paper presented a technique to derive error detectors for protecting an application from data errors. The error detectors are derived automatically using compiler-based static analysis from the backward program slice of critical variables in the program. The slice is optimized aggressively based on specific control-paths in the application, to form a checking expression. At runtime, the executed control path is tracked using specialized hardware and the *checking expressions* corresponding to the control-path are invoked. The checking expression recomputes the value of the critical variable and a mismatch between the recomputed and original values indicates an error.

Acknowledgements

This work was supported in part by the U.S. Department of Commerce under Grant SBAHQ-05-I-0062, NSF grant CRI CNS 05-51665, Gigascale Research Center (GSRC/Marco), Motorola Corporation, and Intel Corporation.

References

[1] D. Evans, et al., LCLint: A tool for using specifications to check code, In Proc. ACM Second Symp. on Foundations of Software Engineering (FSE), 1994, pp. 87-96.

[2] M. D. Ernst, et al., Dynamically discovering likely program invariants to support program evolution, In Proc. Intl. Conf. on Software Engineering (ICSE), 1999, pp. 213-224.

[3] K.Pattabiraman, Z.Kalbarczyk, R.K. Iyer, Application-based metrics for strategic placement of detectors, *11th International Symposium on Pacific Rim Dependable Computing(PRDC)*, 2005, pp. 95-102.

[4] N.S. Oh, P. P. Shirvani, E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors, IEEE Transactions on Reliability, 2002, 51(1), pp. 63-75.

[5] N.S. Oh, S. Mitra, E.J. McCluskey, ED4I: Error Detection by diverse data and duplicated instructions in super-scalar processors, IEEE Transactions on Reliability, 2002, 51(1), pp. 180-199.

[6] Mark Weiser, *Program slicing*, In 5th International Conference on Software Engineering, 1981, pp. 439-449.

[7] C. Lattner and V. Adve. *LLVM: A compilation framework for lifelong program analysis & transformation*. In Proc. ACM Symp. on Code Generation and Optimization (CGO'04), 2004, pp. 75.

[8] Jim Gray. *Why do computers stop and what can be done about it?* In Proc. Fifth Symp. Reliability in Distributed Software and Database Systems, 1986, pp. 3-12.

[9] R. Cytron, et al., *Efficiently computing static single assignment form and the control dependence graph*, ACM Trans. on Programming Languages and Systems 13(4), 1991, pp. 451-490.

[10] M. Carlisle, A. Rogers. Software caching and computation migration in Olden, In 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PpoPP), 1995, pp. 29-38.

[11] N. Nakka, K.Pattabiraman and R.K. Iyer, *Processor Level Selective Replication*, to appear in Proc. Dependable Systems and Networks (DSN), 2007.

[12] N. Nakka, et al., *An architectural framework for providing reliability and security support*, Proc. Intl. Conference on Dependable Systems and Networks (DSN), 2004, pp. 585.

[13] K. Pattabiraman and R.K. Iyer, *Automated Derivation of Application-aware Error Detectors using Compiler Analysis*, UILU-ENG-07-2203, Technical Report, Univ. of Illinois (Urbana-Champaign), Jan. 2007.

[14] G. A. Reis, et al., *SWIFT: Software Implemented Fault Tolerance*, In Proc. 3rd International Symposium on Code Generation and Optimization(CGO), Washington, DC, 2005, pp.243-254.

[15] S. Chandra, P. M. Chen. *How fail-stop are faulty programs?* In Proc. 28th Symposium on Fault-Tolerant Computing (FTCS), 1998, pp. 240.