

Automated Derivation of Application-Aware Error Detectors using Compiler Analysis

Karthik Pattabiraman and Ravishankar K. Iyer

Center for Reliable and High Performance Computing, University of Illinois(Urbana-Champaign)
{pattabir, 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 and constantly monitor the application at runtime. If an error is detected at runtime, the application is stopped so as to prevent error propagation and enable a clean recovery. Experiments show that the derived detectors achieve low-overhead error detection while providing high coverage for errors that matter to the application.

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). Many static analysis [1][2][3] and dynamic analysis [8] [9] 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 subtle software errors (such as timing and synchronization errors) [20] [26] [33], which are not caught by static and dynamic methods.

In order to detect runtime errors, we need mechanisms that can provide high-coverage, low-latency (rapid) error detection to: (i) preempt uncontrolled system crash/hang and (ii) prevent propagation of erroneous data and limit the extent of the (potential) damage. Eliminating error propagation is essential because programs, upon encountering an error that could eventually lead to a crash, may execute for billions of cycles before crashing [27]. During this time, the program can exhibit unpredictable

behavior, such as writing corrupted state to a checkpoint file [32].

Duplication has traditionally been used to provide high-coverage at runtime for software errors and hardware-errors. However, duplication suffers from the following disadvantage: 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. IBM G5 processors perform the comparison in hardware to reduce the performance overhead of duplication, but have high hardware design complexity. Further, duplication detects many benign errors [16]. Benign errors are errors that would not have impacted the application even if the error had not been detected.

The approach presented in this paper derives error detectors (or checks) based on static analysis but performs the detection at runtime. It takes into account the placement of checks to preempt crashes and provides high-coverage to detect errors that result in application failures. The approach is complementary to existing static analysis techniques and detects subtle errors such as timing errors in the program. In addition, the derived checks can naturally detect hardware errors that occur in the processor and the memory. The main contribution of this paper is that it extends static analysis techniques to derive runtime error detectors based on application properties.

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

- Derived detectors detect around 75% of errors that propagate and cause crashes, and in many cases (80% of detections), detects the error before propagation.
- The derived detectors detect only 2% of the benign errors. In comparison, full-duplication detects 40-50% of benign errors [16].
- The average performance overhead of the derived detectors across 14 benchmark applications is 33%.

2. Related Work

This section considers related work on locating software bugs using derived invariants as well as on runtime detection of hardware and software errors. The set of techniques discussed can be divided into six broad groups as follows:

Static Analysis Techniques: There have been many techniques proposed to find bugs in programs based on a static analysis of the application code [1][2][3]. These techniques validate the program based on a well-understood fault model, usually specified based on common

programming bugs (e.g. NULL pointer dereferences). The techniques attempt to locate errors across all feasible paths in the program (a program path that corresponds to an actual execution of the program). Determining feasible paths is known to be an impossible problem in the general case. Therefore, these techniques make approximations that result in the creation of spurious paths. This in turn can result in the approach finding errors that will never occur in a real execution, leading to wasteful detections.

Dynamic Invariant Deduction: These techniques derive code-specific invariants based on dynamic characteristics of the application. An example of this technique is DAIKON [8], which derives code invariants such as the constancy of a variable, linear relationships among sets of program variables and inequalities involving two or more program variables. DAIKON's primary purpose is to present the invariants found to programmers, who can validate them based on their mental model of the application. The invariants are derived based on the execution of the application with a representative set of inputs, called the training set. Inputs that are not in this set may result in the invariants being violated even when there is no error in the application.

DIDUCE [9] is a dynamic invariant detection approach that uses the invariants learned during an early phase of the program to detect errors in the subsequent execution. It is unclear how the invariants learned during the early stages of execution well represent the entire application's execution.

Rule-based Error Detectors: Hiller et al [5] provide rule-based templates to the programmer for specifying runtime error detectors. However, the programmer needs to choose the right templates to be used for detection and supply the parameters of the template to the system. The technique automates the placement of the derived detectors to preempt application failures [6]. Pattabiraman et al. [10] derive error detectors based on rule-based templates, wherein the choice of templates and the parameters to be checked are automatically derived. The generic problem with rule-based detectors however, is that they are specific to an application domain (e.g. specific embedded applications), and it is difficult to make them work for general-purpose applications.

Inferring specifications from code: The main idea here is to learn program patterns from source code analysis and consider violations of these patterns as program bugs [4]. Patterns are learned from localized code samples and extended to the whole code base. The techniques are useful for finding common programming errors such as copy-and-paste errors [12] or an error due to the programmer forgetting to perform an operation [4], such as releasing locks. It is unclear if they can be used for detecting more subtle errors that occur in well-tested code, such as timing and memory errors, as these errors may not be easily localized to particular code sections [20]. Further, these techniques have large false-positive rates i.e. many errors do not correspond to real bugs.

Compiler-based Replication: The entire program is replicated either at the source-level [13], instruction level [14] or at the compiler intermediate code level [15]. The

results of the replicated instructions (statements) are compared after every instruction (statement) [13] or at selected program points such as stores to memory [14], [15]. These techniques require the entire program to be replicated in order to provide protection, which can result in high performance overheads (90-100%)¹. An important issue in all low-level replication techniques is that they result in the detection of many errors that have no impact on the application (benign errors) [27]. This constitutes a wasteful detection (and consequent recovery) from the application's viewpoint.

Further, duplication-based techniques offer limited protection from software faults and permanent hardware faults because the original program and the duplicated program can suffer from common mode errors. In order to deal with common-mode errors, *diverse execution techniques* that execute two different versions of the same program and compare the results must be used.

ED4I [17] is a software-based diverse execution technique to protect against transient and permanent hardware faults. The original program is transformed into a different program in which each data operand is multiplied by a constant value k . The original program and the transformed program are both executed on the same processor and the results are compared. Since the transformed program operates on a different set of data operands than the original program, it is able to mask hardware errors in processor functional units and memory. However, the technique cannot detect software errors that result in incorrect computation of data values in both the original program and the transformed program.

Runtime Verification: Runtime-verification techniques attempt to bridge the gap between formal techniques such as model checking and runtime checking techniques. These techniques check whether the program violates a programmer-specified safety property [18][19] by constructing a model of the program and checking the model based on the actual program execution. The checking is done at specific program points depending on the model. However, if there is a general error in the program there is no guarantee that the program will reach the check before crashing. Since the papers describing these techniques [18][19] only consider errors that are directly detectable (by the checking technique), it is unclear if the techniques provide useful runtime coverage for a random hardware or software error.

Runtime-error-detection techniques: The static techniques we have discussed are geared towards detecting errors at compile-time, while the dynamic analysis techniques are geared towards providing feedback to the programmer. Both these types are *fault-avoidance* techniques (fault is removed before the program is operational).

Despite the existence of these techniques and rigorous program testing, subtle but important errors such as timing errors persist in a program [20] (as these errors cannot be

¹ Most of these studies consider replication on RISC processors, which have excess capacity in terms of registers. A recent study on instruction replication for x86 processors reports an average overhead of 900% [29]

detected at compile-time). *Runtime-error detection* techniques are geared towards addressing these errors (and also hardware errors). As we have already seen, full replication can detect many of these errors; but not only does it incur significant performance overheads, it also results in a large number of benign error detections that have no impact on the application [16]. Thus, there is a need for a technique that takes advantage of application characteristics and detects arbitrary errors at runtime without incurring the overheads of replication.

The question that we attempt to answer in this paper is: Is it possible to derive software-based runtime checks to minimize the detection latency, preempt crashes and reduce fail-silent violations? This is crucial in order to perform rapid recovery upon application failure [26].

3. Approach

This section presents an overview of the proposed detector derivation approach. The approach is based on the well-known 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 [21]. Slicing techniques can be classified into static and dynamic slicing techniques [22].

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. Checking expressions are referred to synonymously as checks in the paper. Checks are placed after the computation of the critical variable in the original program.

Detector: Each derived path can have its own checking expression. At runtime, the program is monitored and only the checking expression corresponding to the runtime path followed is asserted. A detector is defined as the combination of the checking expression and the runtime monitoring.

3.2 Slicing Algorithm

The slicing algorithm presented in this paper is a static slicing technique that considers all possible dependences between instructions in the program regardless of program inputs. It does not perform inter-procedural slicing allowing the analysis to be scaled to large applications. This can affect the coverage of the derived detectors.

However, by placing multiple detectors in the program at critical variables, it is possible to achieve high coverage (as shown in our results in Section 5.2). This is because at least one of the detectors placed in the program will be able to detect the error.

3.3 Steps in Detector Derivation

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

Identification of critical variables: The critical variables are identified based on an analysis of the dynamic dependence graph of the program presented in [10]. 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.

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.

Check derivation, Check insertion and 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 path.

Runtime checking in hardware and software:

The control path followed is tracked by the inserted instrumentation in hardware at runtime (*path-tracking*). 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. Otherwise, execution continues normally.

3.4 Example of Derived Detectors

The derived detectors are illustrated using a simplified example of an *if-then-else* statement in Figure 1. A more realistic example is presented in Section 4. In the figure, the original code is shown in the left and the checking code added is shown in the right. Assume that the detector placement analysis procedure has identified f as one of the critical variables that need to be checked before its use in the following basic block. For simplicity, only the instructions in the backward slice of variable f are shown in Figure 1.

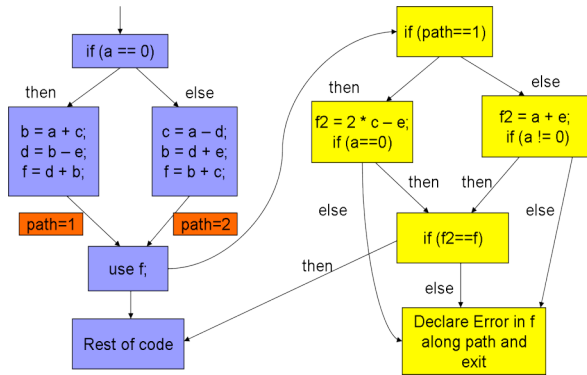


Figure 1: Example code fragment with detectors inserted

There are two paths in the program slice of f , corresponding to each of the two branches. The instructions on each path can be optimized to yield a concise expression that checks the value of f along that path (shown in yellow in Figure 1). In the case of the first path ($path=1$), the expression reduces to $(2 * c - e)$ and this is assigned to the temporary variable $f2$. Similarly the expression for the second path ($path=2$) corresponding to the *else* branch statement reduces to $(a + e)$ and is also assigned to $f2$. Instrumentation is added to keep track of paths at runtime.

At runtime, when control reaches the use of the variable f , the correct checking expression for f is chosen based on the value of the $path$ variable and the value of $f2$ is compared with the value of f computed by the original program. In case there is a mismatch, an error is declared and the program is stopped.

3.5 Hardware/Software Implementation

In the proposed technique, the analysis of the program, derivation of the checking expression and the addition of instrumentation is entirely done at compile-time. At runtime, the added instrumentation keeps track of the path followed and executes the checking expression corresponding to the path. While the runtime checking can be performed in hardware or software, we provide a combined hardware-software implementation in this paper.

There are two sources of runtime overhead for the detector: (1) the overhead of keeping track of the control path followed and (2) the overhead of executing the check.

Path Tracking: The overhead of tracking paths is significant (4x) when done in software². Therefore, a prototype implementation of path tracking is presented in hardware. This hardware is integrated with the Reliability and Security Engine (RSE) [7]. 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 level. The path-tracking hardware is implemented as a module in the RSE framework³ and is configured at application load-time. The monitoring is done in parallel with the main program, thereby reducing the performance overhead of the monitoring.

² As measured from an experimental evaluation of our earlier implementation of this technique in software alone

³ Generically, it can be implemented on any FPGA.

In this paper, the behavior of the path-tracking module is simulated in software and the conceptual design of the hardware module is presented in Section 4.4.

Checking: In order to further reduce the performance overhead, the check execution itself can be moved to hardware. This would involve compiling the checking expressions directly to hardware and implementing them in the RSE. In our current implementation, the checking is done in software.

3.6 Discussion

As illustrated in the example in Figure 1, a checking expression consists of only those instructions that are part of the computation of the critical variable and is specific to each control path that reaches the variable from the top of the function. Since, the specific set of instructions is optimized separately from the rest of the program, the check introduces a level of diversity in the recomputation of the critical variable. This diversity is valuable in that it provides the detection of errors in the program instructions that are interleaved with the critical variable’s computation.

Assume that an error in the interleaved instructions affects one of the instructions involved in the computation of the critical variable (i.e. instructions in the slice). The representation of this instruction in the checking expression will likely be different from the representation in the original program. For example, an *add* instruction in the original program may be optimized to a simple *mov* instruction in the checking expression (assuming that one of the operands of the add instruction was zero for the path considered). Therefore, the probability of the error impacting the value produced by the instruction in both the original program and in the checking expression is small. *Hence, common mode errors between the checking expression and the original program can be reduced.*

3.7 Errors Detected

The fault model in this study covers *errors* in data values due to both hardware and software faults:

Hardware fault: Any transient error in the following hardware components:

- *Processor data path:* Includes errors in functional units or in the register file that result in data-value corruption.
- *Processor control path:* Includes errors in the instruction decode and issue units that result in the wrong instruction being executed.
- *Memory/Cache:* Errors in the memory or cache caused due to cosmic radiation or electric disturbances. These errors will also be detected by techniques such as ECC, if these techniques are deployed in memory and cache.

The checking expressions derived by the proposed technique can detect the above three categories of hardware errors provided they affect the computation of the critical variables in either the program or the check, but not in both the program and the check.

Software fault: Any program defect that causes transient data-value corruptions such as:

- Synchronization errors or race conditions that result in corruptions of data values due to incorrect sequencing of operations.

- Memory corruption errors, e.g., buffer-overflows and dangling pointer references that cause arbitrary data values to be overwritten in memory, use of uninitialized or incorrectly initialized values. These errors could result in the use of unpredictable outcomes depending on the platform and the environment.

The checking expressions derived can detect the above software errors, provided the error occurs in the code interleaved with the computation of the critical variable, *and*, the error affects one of the values involved in the computation of the critical variable in the original program. The assumption is that the software error does not occur during the recomputation of the critical variable in the checking expression⁴, due to the diversity introduced in the checking expression (see Section 3.6).

3.8 Errors not detected

Examples of errors that would not be detected are as follows:

- **Hardware Errors:** Permanent or persistent errors in some hardware components would not be detected. For example, errors that persist in the functional units of the processor would not be detected if both the original program statement and the checking expression were computed by the same functional unit. However, the probability of this occurring is small as modern microprocessors have excess capacity of functional units.
- **Software Errors:** A common program input that results in the program computing the wrong result, although the result is valid according to the code semantics. Similarly errors that arise due to the code not being faithful to the program semantics would not be detected. This is because the detectors are derived from the code and not from the specifications.

4. Detector Derivation Technique

This section illustrates the detector derivation methodology using a real example of a bubble-sort program shown in Figure 2a. The actual analysis and transformations are carried out on the compiler’s intermediate representation of the C code, which is similar to assembly code.

We use the LLVM compiler [23] for the analysis and derivation of error detectors. The derivation of detectors is done by the introduction of a new pass into LLVM. This allows the use of LLVM features while preserving the benefits of the new analysis introduced. A simplified version of the LLVM intermediate code corresponding to the inner-loop in the bubble-sort program is shown in Figure 2b.

LLVM and SSA Form: LLVM is an optimizing compiler infrastructure that uses Static Single Assignment form (SSA) [24] as its intermediate code representation. In deriving the backward program slice, two well understood properties of SSA form are used as follows:

In SSA form, each variable (value) is defined exactly once in the program, and the definition is assigned a unique name [24]. This unique name makes it easy to identify data dependences among instructions.

SSA form uses a special static construct called the *phi* instruction that is used to keep track of the data dependences when there is a merging of data values from different control edges. The *phi* instruction includes the variable name for each control edge that is merged and the corresponding basic block. This instruction allows the detector derivation technique to specialize the backward slice based on control-paths.

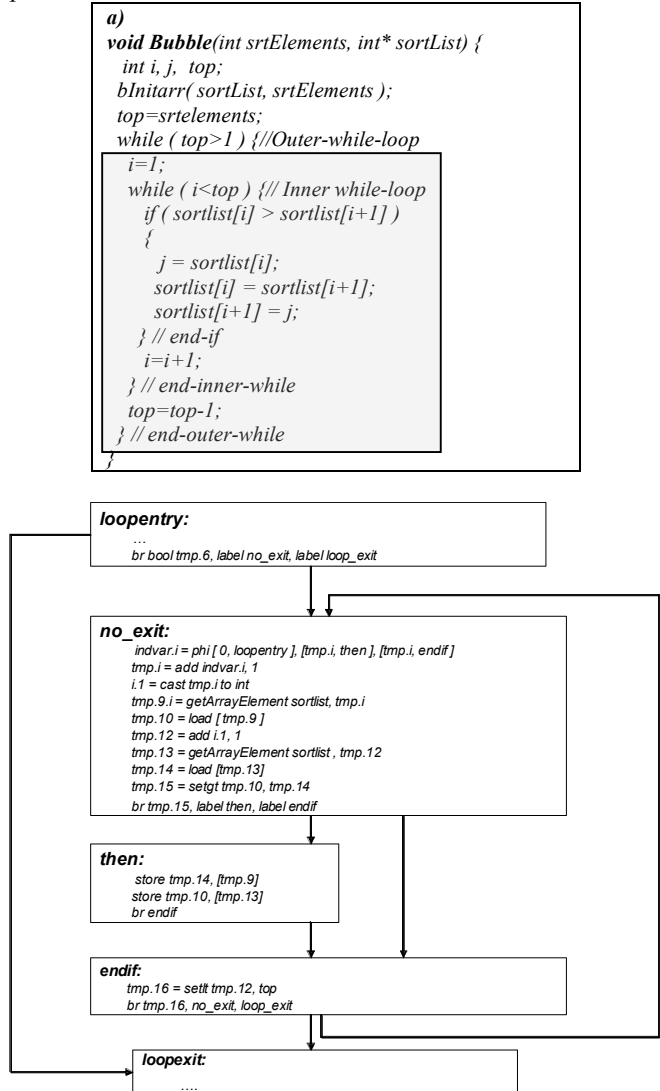


Figure 2: (a) Example code fragment (bubble-sort) and (b) and the corresponding LLVM intermediate code for the inner while loop in (a)

Value Recomputation Pass: In order to derive checking expressions, a new compiler pass called the *Value Recomputation Pass* was introduced into the LLVM compiler. The input to the new pass is the LLVM intermediate code of the original program, and its output is LLVM intermediate code with the checks inserted.

The Value Recomputation pass 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.

⁴ This is a reasonable assumption as software errors in programs disappear upon recomputation [33]

The output of the pass is provided as input to other optimization passes in LLVM, which in turn are used to derive the checking expression. *By extracting the path-specific backward slice and exposing them to other optimization passes in the compiler, the Value Recomputation pass enables aggressive compiler optimizations to be performed on the slice than otherwise possible.*

An important contribution of this paper is the algorithm used for creating the path-specific slice for critical variables. The notion of a path-specific slice has been used in model checking of C programs for errors, to improve accuracy in locating the error-causing statements in the program [25]. While the algorithm presented in [25] greatly reduces the number of paths considered for error checking, it is still prone to the spurious path problem. Further, the algorithm cannot be extended to deriving runtime error detectors in a straightforward manner.

To the best of our knowledge, this paper presents the first path-specific static slicing algorithm for deriving runtime error detectors.

4.1 Identification of critical variables

The identification of critical variables is performed based on the technique proposed in [10]. The technique considers potential error propagation in the program based on its Dynamic Dependence Graph (DDG). Critical variables are chosen based on heuristics computed from the DDG. Critical variables are defined as those variables having the highest fanouts (number of dynamic uses) in a function. Placing detectors at critical variables provides the maximum error detection coverage for errors that result in program crashes and fail-silent violations [10].

Computation of Fanouts: The compiler first instruments the program and the instrumented program is executed (on representative inputs) to gather profile data. The profile data is then given as input to the *Value Recomputation* which computes the fanout of a variable as the sum of the execution frequencies of the basic blocks containing static uses of the variable. The pass then chooses the top N variables with the highest fanouts in each function as critical variables (for that function).

Assume that the critical variable chosen for the example in Figure 2a is *sortlist[i]*. The intermediate code representation of this variable is the instruction *tmp.10* in Figure 2b.

4.2 Computation of backward slice of critical variable

This section presents the algorithm to compute the backward slice of a critical variable for each control path from the beginning of the function to the program location that performs the computation of the critical variable.

Overview of 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).

The rationale for each of these cases is presented below:

- *Intra-procedural Slices:* As already mentioned, 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 2a, 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. This second check ensures that the variable within the loop is computed correctly and hence the variable can be used directly in the check.
- *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 across multiple iterations and one for the critical variable.

Algorithm Description. The pseudo-code of the algorithm is shown in Figure 4. The algorithm maintains the list of instructions in the slice specialized for each path in the array *SliceList*. The function *computeSlice* takes as input the critical instruction and outputs the *SliceList* array, which contains for each acyclic path in the control-flow graph, the instructions in the slice corresponding to that path (in execution order). When the *ComputeSlice* function terminates, the following properties hold true: (1) The backward slice of the critical instruction along each acyclic control path in the function has been computed and (2) Checks have been added for variables that affect the critical instruction but could not be included in the slice of the critical instruction (for the reasons considered above).

```

Function visit( seedInstruction, pathID, parent ):
ActiveSet = { seedInstruction }
if parent == 0:
    SliceList[ pathID ] = { }
else:
    SliceList[ pathID ] = SliceList[ parent ]
nextPathID = pathID
while not empty( ActiveSet ):
    I = Remove instruction for ActiveSet
    Visited[ BasicBlock(I) ] = true
    // Do not consider interprocedural slices
    if I is a function argument or constant:
        terminal = true
    else if I is a non-phi instruction:
        SliceList[ pathID ] = SliceList[ PathID ]
            U { I }
        ActiveSet = ActiveSet U operands( I )
    else if I is a phi instruction:
        for each operand of the phi:
            // Check if a loop is encountered
            // or if going back multiple iterations
            if not ( Visited [ BasicBlock(operand) ]
                and not CrossingInsn(I, operand) )
                nextPathID = pathID + 1
                result = call Visit( operand,
                    nextPathID, pathID )
                terminal = terminal OR ~(result)
            else:
                SeedList = SeedList U { operand }
        // Add the path to the pathList if terminal path
        if (terminal)
            PathList = PathList U { pathID }
    return terminal

Function computeSlices (criticalInstruction):
SeedList = { criticalInstruction }
PathList = { }
while not empty( SeedList ):
    seedInstruction = Remove instruction from SeedList
    call visit( seedInstruction, 0, 0 )
return PathList, SliceList

```

Figure 3: Pseudo-code for path-specific program slicing starting from critical instruction

The actual traversal of the dependence graph occurs in the function *visit*, which takes as input the starting instruction, an ID (number) corresponding to the control-flow path it traverses (index of the path in the *SliceList* array), and the index of the parent path.

The *visit* function visits each operand of an instruction in turn, adding them to the *SliceList* of the current path. When a *phi* instruction is encountered, it spawns a new path for each operand of the *phi* instruction (by calling the *visit* function recursively on the operand) with a new path ID and the current path as the parent. The traversal is then continued along this new path.

Only terminal paths are added to the final list of paths (*PathList*) returned by the *ComputeSlice* procedure. A terminal path is defined as one that terminates without spawning any new paths.

Some instructions cannot be recomputed in the checking expression, because performing recomputation of such instructions can alter the semantics of the program. Examples are *mallocs*, *frees*, function calls and function

returns. Omitting *mallocs* and *frees* does not seem to impact coverage except for allocation intensive programs, as shown by our results in Section 6.2. Omitting function calls and returns does not impact coverage for program functions because the detector placement analysis considers each function separately for identifying critical variables.

The algorithm also takes into account certain features of SSA form to optimize its performance (not shown in Figure 3). For example, SSA form ensures that each definition of a variable dominates its use [35] and the algorithm uses this to avoid re-traversing some paths. Further, the algorithm performs routine optimizations such as memoization for efficiency.

Output of algorithm. The backward slice of *tmp.10* consists of two paths shown in Figure 4. The first path in Figure 4 (path 0) corresponds to the control-flow transfer from the basic block *no_exit* to the basic block *looptentry*, whereas the second path (path 1) corresponds to the control-flow transfer from the basic block *endif* to the basic block *looptentry.1.i*.

Memory Dependences. 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 does not have to be constrained by the actual number of physical registers available for the machine.

<p>Path 0: no_exit → looptentry</p> <pre> indvar.i = phi [0, looptentry], [tmp.i, then], [tmp.i, endif] tmp.i = add indvar.i, 1 tmp.9 = getArrayIndex sortlist, tmp.i tmp.10 = load [tmp.9] </pre>	<p>Path 1: endif → looptentry</p> <pre> indvar.i = phi [0, looptentry], [tmp.i, then], [tmp.i, endif] tmp.i = add indvar.i, 1 tmp.9 = getArrayIndex sortlist, tmp.i tmp.10.i = load [tmp.9] </pre>
---	---

Figure 4: Path-specific slices extracted for the example

There are cases however, when it may not be possible to promote a memory objects to a register e.g. pointer references to dynamically allocated data. In such cases, the Value Recomputation pass duplicates the load of the memory object, provided the load address is not modified along the path considered from the load instruction to the critical instruction. While duplicating loads may seem to introduce a performance bottleneck in the program, it would not matter much in practice as the compiler may assign the loaded value to a register (and remove both the original and duplicate loads). Even if the load operand is from a memory operand, it is likely to hit in the cache due to the fact that is the value was loaded in by the original program before the check is executed.

It is also possible to duplicate the store instruction that caused the memory dependence. However, duplicating loads seems to be sufficient to obtain high coverage even for pointer-intensive applications involving dynamic data-structures (as seen in Section 5.3).

4.3 Check Derivation, Check Insertion and Instrumentation

Check Derivation: The Value Recomputation pass places the instructions in the backward slice of the critical variable

corresponding to each control path in a separate basic block. The pass also replaces the *phi* instructions in the slice with the incoming value corresponding to the control edges along each path. This allows the compiler to substitute the incoming value directly in the recomputation of instructions that originally used the phi-value. Further, since there are no control-transfers within the sequence of instructions included in each path, the compiler is able to optimize the instruction sequence for the path much more aggressively than it would have optimized otherwise. This is because the compiler in and of itself, would not consider specific control paths (in the slice) when performing optimizations.

Check Insertion. After the check is derived, the Value Recomputation pass does the following:

1. Inserts the check consisting of the reduced expressions from the slice immediately after the computation of the critical instruction (in Figure 5, it places the check after the *tmp.10*).
2. Creates copies of variables used in the checking expression that are not live at the check insertion point. For example in Figure 5, a copy of *tmp.i* is created, as the value of *tmp.i* is overwritten in the loop before the check can be reached. The copy is used in the check.
3. Renames the values of the variables in the check to avoid conflicts with the original program.
4. Inserts a branch instruction to execute the correct checking expression based on *pathVal*, which stores the control-path retrieved from the hardware path-tracking module (Section 4.4).

The LLVM intermediate code from Figure 2b with the checks inserted by the Value Recomputation pass is shown in Figure 5.

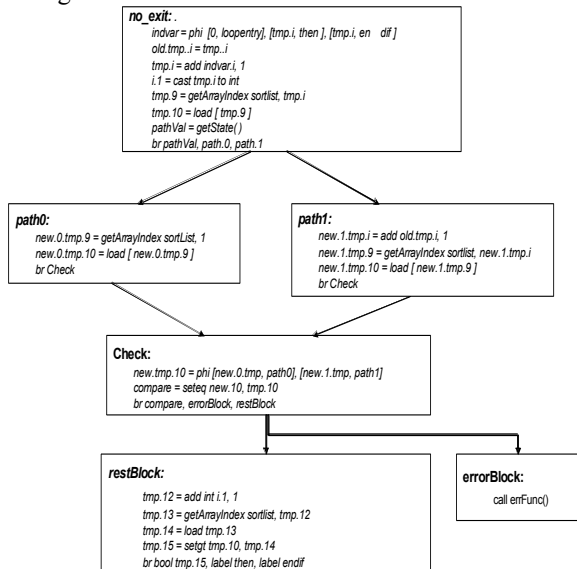


Figure 5: LLVM Intermediate code with inserted checks

Instrumentation: The pass instruments selected control-flow edges that uniquely identify the paths extracted by the backward slicing algorithm. This instrumentation interfaces with the hardware module to track paths at runtime (see Section 4.4).

4.4 Discussion

As illustrated in the example in Figure 5, the instructions in the checking expression are optimized separately from the rest of the program. *As a result, the check introduces a level of diversity in the recomputation of the critical variable.* This diversity provides detection of errors in the instructions involved in the critical variable’s computation.

Consider what happens when an error affects an instruction that is involved in the computation of the critical variable. Assume that the error affects the instruction that computes *tmp.i* in Figure 2b (this instruction indirectly impacts the computation of the critical variable *tmp.10*).

We now describe how this error is detected by the checking expressions in *path0* and *path1*, when the corresponding control paths are executed by the program.

First consider the case when the runtime path followed corresponds to the execution of the checking expression in the basic block *path0* (Figure 5). In *path0*, the compiler performs constant propagation and replaces the computation of *tmp.i* with the constant *1* in Figure 5. As a result, the error in the computation of *tmp.i* is not manifested in *path0*. Hence, the value of the critical variable computed in *path0*, namely *new.0.tmp.10*, is different from the value of the critical variable computed in the original program (Figure 5). Therefore the error in the computation of *tmp.i* is detected along *path0*.

Now consider the case when the path followed corresponds to the execution of the checking expression in *path1* (Figure 5). The Value Recomputation Pass inserts code to copy the original value of *tmp.i* into *old.tmp.i* before *tmp.i* is overwritten in the program. The value *old.tmp.i* is used in the checking expression in *path1* to recompute the value of *tmp.i*, namely *new.1.tmp.i*, which in turn is used to recompute the critical variable in *path1*. The value *new.tmp.i* is computed and stored separately from the original value *tmp.i*, and consequently does not suffer from the error that affected the computation of *tmp.i*. As a result, the value of the critical variable computed in *path1*, namely *new.1.tmp.i* is different from the one computed in the original program (Figure 5). Therefore the error in the computation of *tmp.i* is detected along *path1*.

In the first case, the checking expression performed a recomputation of the critical variable with diversity in instructions (*path0*) while in the second case it performed the recomputation with diversity in data (*path1*). In both cases, the diversity was introduced by the transformations carried out by the Value Recomputation Pass and subsequent optimization passes. *Therefore, the diversity introduced by the checking expressions allows the detection of errors that may not have been detected due to simple duplication alone.*

4.5 Implementation of Path-tracking in Hardware

The path-tracking hardware keeps track of the control paths encoded as finite state machines. The *Value Recomputation pass* synthesizes the state machines for each check automatically from the program. The algorithm to convert the control-flow paths corresponding to each check into state machines is straightforward and is not described here.

As explained in Section 3.5, the path-tracking hardware is implemented as a module in the RSE and monitors the main processor data path. The state machines corresponding to each check in the application are programmed into the path-tracking module at application load time. Since the number of checks is small (10-50), the memory required to store the state machine in hardware is small (Section 5.4).

Other approaches: Many approaches have been proposed for profiling application control paths using specialized hardware [31]. The goal of these approaches is to create statistical aggregates of application behavior, rather than track specific paths. Zhang et al. [30] propose a hardware module that interfaces with the processor pipeline to track paths for detecting security attacks. However, their approach requires every branch in the program to be instrumented, which can lead to prohibitive overheads. Our approach is aimed at tracking specific control-paths in the program (for which checks are derived), and requires only selected control edges (branches) to be instrumented.

Interface with main processor: The main processor uses special instructions called CHECK instructions to invoke the RSE modules. The path tracking module supports three primitive operations encoded as CHECK instructions. The operations are as follows:

emitEdge(from, to): Triggers transitions in the state machines corresponding to one or more checks. Each basic block in the program is assigned a unique identifier assigned by the Value Recomputation pass. This operation indicates that control is transferred from the basic block with identifier *from* to the basic block with identifier *to*.

getState(checkID): Returns the current state of the state machine corresponding to the check, and is invoked just before the execution of the check in the program.

resetState(checkID): Resets the state-machine for the check given by *checkID*. This operation is invoked after the execution of the check in the program.

Module Components: The path-tracking module is shown in Figure 6. It consists of three main components as follows:

Edge Table: Stores the mapping from control-flow edges to edge-identifiers for instrumented edges in the program. Each instrumented control-flow edge is assigned a unique index and is mapped to the identifiers assigned to the source and sink basic blocks for that edge.

State Vector: Holds the current state of the state machine corresponding to the checks, with one entry for each check inserted in the program.

State Transition Table: Contains the transitions corresponding to the state machines. The rows of the state transition table correspond to the edge indices, while the columns correspond to the checks. The cells of the table contain the transitions that are fired for each check when a control-flow edge is taken in the program.

RSE Interface: Converts the CHECK instructions from the main processor into signals specific to the path-tracking module⁵. Similarly, converts signals from the path-tracking module into special flags in the main processor. This is a common component shared by all modules of the RSE

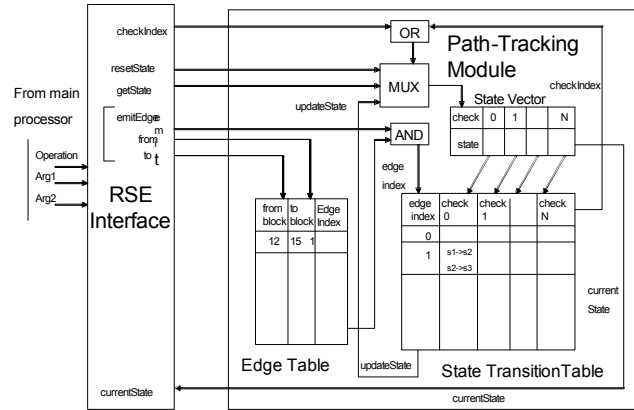


Figure 6: Hardware module for tracking paths

Module Operation: The operation of the path-tracking module for each of the primitive operations (executed in the main processor) is considered below:

CHECK instruction with emitEdge operation is executed in the main processor:

- RSE interface asserts the *emitEdge* signal and sends the basic block identifiers that constitute the edge in the *from* and *to* lines.
- The *from* and *to* identifiers are looked up in the *edge table* and the edge index corresponding to the edge is sent to the *state transition table*.
- The row corresponding to the edge is looked up in the *state transition table*.
- For each non-empty table-entry in the column corresponding to the checks, the states in the LHS of the transitions stored in the table entry are compared to the current state of the check in the *state vector*.
- If the states match, then the transition is fired and the state vector entry corresponding to the check is updated with the state in the RHS of the transition that matched.

CHECK instruction with the getState operation is executed in the main processor:

- RSE interface asserts the *getState* signal and sends the identifier of the check on the *checkID* line to the path-tracking module.
- The path tracking module looks up the state in the *state vector* and sends it to the RSE interface through the *currentState* line. This in turn is sent to the main processor and is returned as the value of the CHECK instruction (through a special register in the RSE).

CHECK instruction with the resetState operation is executed in the main processor: This is similar to the *getState* operation, but no value is returned to the RSE interface.

Function calls/returns: The *state vector* needs to be preserved across function calls and returns. This is done by pushing the *state vector* on a separate stack (different from the function call stack) along with the return address upon a function call and by popping the stack upon a return. The Value Recomputation pass generates code that uses special CHECK instructions to manipulate the stack on function calls/returns. The details are omitted due to space constraints.

⁵ This is done by tapping the *Fetch_out* signal from the main pipeline.

5. Experimental Setup

This section describes the mechanisms for measurement of performance and coverage provided by the proposed technique. It also describes the benchmarks used for evaluation.

5.1 Performance Measurements

All experiments are carried out on a single processor P4 machine with 1GB RAM and 2.0 Ghz clock speed running on the Linux operating system. The performance overheads of each individual component introduced by the proposed technique can be measured as follows:

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

The performance overhead of the hardware module is likely to be very small⁶ because the path tracking can be done in parallel with the execution of the main program. The path-tracking module needs to be synchronized with the main processor only at the *getState* operation, and can execute asynchronously the rest of the time. This allows the path-tracking module itself to be organized as a sequence of pipeline stages for efficiency. Hence, the overhead of path-tracking is not considered in measuring performance overheads (as path tracking is done in hardware).

5.2 Coverage Measurements

Fault Injections: Faults are injected into the application code to measure the coverage of the 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. The uses of the program variable in the original program are substituted with the return value of 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) the program may terminate by taking an exception (crash), (2) the program may continue and produce correct output (success), (3) the program may continue and produce incorrect output (fail-silent violation) or (4) the program may timeout (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, after which the program crashes. Hence, when a detector detects a crash, it is in reality, preempting the crash.

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 such 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.

5.3 Benchmarks

Table 1: Characteristics of Benchmark programs

Bench mark	Lines of C code	Description of program
IntMM	159	Matrix multiplication of integers
RealMM	161	Matrix multiplication of floating-points
FFT	270	Computes Fast-Fourier Transform
Quicksort	174	Sorts a list of numbers using quicksort
Bubblesort	171	Sorts a list of numbers using bubblesort
Treesort	187	Sorts a list of numbers using treesort
Perm	169	Computes all permutations of a string
Queens	188	Solves the N-Queens problem
Towers	218	Solves the Towers of Hanoi problem
Health	409	Discrete-event simulation using double linked lists
Em3d	639	Electro-magnetic wave propagation in 3D (using single linked lists)
Mst	389	Computes minimum spanning tree (graphs)
Barnes-Hut	1427	Solves N-body force computation problem using octrees
Tsp	572	Solves traveling salesman problem using binary trees

In order to evaluate the system, 9 programs from the Stanford benchmark suite and 5 programs from the Olden benchmark suite [28] are used. The former benchmark set consists of small programs performing a multitude of common tasks. The latter benchmark set consists of pointer-intensive programs commonly used to test memory system performance. Table 1 shows a description of the program characteristics. The first nine programs belong to the Stanford suite and the remaining five to the Olden suite.

5.4 Hardware Area overhead

The area overheads for the hardware module are dominated by the three main components of the module presented in Section 4.4. The other components are mainly glue combinational logic and occupy negligible area.

Table 2 presents the formulas used in estimating the size of the dominant hardware components. The size of each of these components depends on (1) the number of control-flow edges corresponding to state transitions (m), (2) the number of checks that must be tracked for the application (n) and, (3) the maximum number of transitions in each entry of the

⁶ While we have not quantified the performance overhead in this paper, our earlier experience with building error detectors in hardware [11] indicates that the performance overhead is less than 5%.

state transition table because the table must be big enough to hold the biggest entry (k).

Table 2: Formulas for calculation of hardware area overheads

Hardware Component	Size (bits)	Explanation
Edge Table	$m * 16 * 3$	Each entry has 3 fields <i>from</i> , <i>to</i> and <i>edgeIndex</i> . Each of these fields consists of 16 bits.
State Vector	$n * 8$	Each entry of the state vector consists of 8 bits, which is the number of bits used to encode states
Transition Table	$n * m * k * 16$	Each state transition consists of two 8-bit fields to encode the starting and ending states of the transition.

6. Results

This section presents the performance (Section 6.1), and coverage results (Section 6.2) 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. The results for the hardware area overheads are presented in Section 6.3.

6.1 Performance Results

The performance results are shown in Figure 7 when 5 critical variables are chosen per function. 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. This can be avoided by implementing the check entirely in hardware, thereby causing minimum perturbation of the program’s behavior. This is a direction for future work.

6.2 Coverage Results

The coverage results (reported in percentages) when 5 critical variables are chosen in each function are reported in Table 3. 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 the *IntMM* application in the 1000 fault injection runs.

Only program crashes that exhibit error propagation (before the crash) 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 3 are summarized as follows:

- 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 number of benign errors detected is 2.5% on average. These errors have no effect on the execution of the application.
- The coverage for fail-silent violations is 41% (on average), and the coverage for hangs is 35%

Table 3: Coverage results for 5 critical variables per function

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

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 easily overcome by placing detectors inside library functions (by the library developer, for instance).

6.3 Hardware Overheads

The results estimating the hardware size overhead across applications are presented in Table 4. The average number of bits stored by the hardware module is 4928. This corresponds to less than 1 KB of storage space in the hardware. The application exhibiting the worst-case overhead (Barnes-hut) occupies 33452 bits, corresponding to less than 4KB of memory. This fits into a standard FPGA BRAM cell which has about 5096KB of memory available[34].

6.4 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 [14][15]. In this optimized mode of execution, the coverage provided by full-duplication is less than 100%. The papers that describe these techniques do not quantify the detection coverage in terms of error propagation, so a direct comparison with by our technique is not possible. Further, the performance overhead of the technique is only 33 %, compared to full-duplication, which incurs an overhead of 60-100% when performed in software [14][15]. An important aspect of the technique is that it detects just 2.5 % of benign errors in an application.

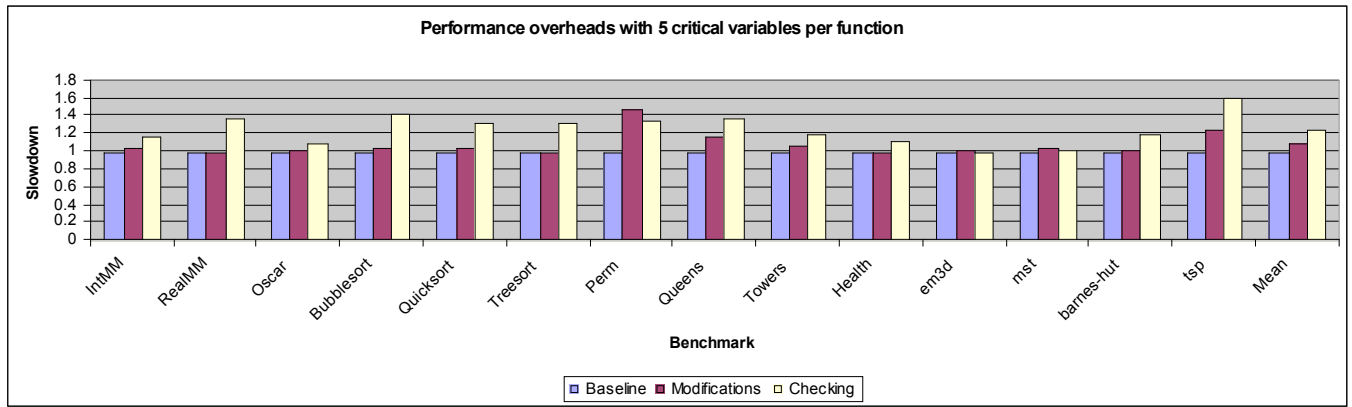


Figure 7: Performance overhead when 5 critical variables are chosen in each function

Table 4: Hardware Size overheads across applications

Application Name	Number of edges (m)	Number of checks (n)	Maximum transitions /entry (k)	Edge Table Size (bits)	Transition table size (bits)	State Vector Size (bits)	Total Size (bits)
IntMM	10	21	3	480	630	168	1278
RealMM	10	21	3	480	630	168	1278
FFT	17	30	4	816	2040	240	3096
Quicksort	19	29	5	912	2755	232	3899
Bubblesort	5	11	1	240	55	88	383
Treesort	10	20	4	480	800	160	1440
Perm	16	27	1	768	432	216	1416
Queens	5	20	1	240	100	160	500
Towers	11	31	1	528	341	248	1117
Health	9	52	1	432	468	416	1316
Em3d	8	30	3	384	720	240	1344
Mst	17	33	10	816	5610	264	6690
Perimeter	26	57	9	1248	13338	456	15042
Barnes-Hut	43	118	6	2064	30444	944	33452
Tsp	9	48	2	432	864	384	1680
Average	14	37	4	432	688	3949	4928

7. Conclusions and Future Work

This paper presented a technique to error detectors for protecting an application from data errors (both due to hardware and software). The error detectors were derived automatically using compiler-based static analysis from the backward program slice of critical variables in the program. The slice is optimized aggressively and differently (from the rest of the code) based on specific control-paths in the application, to form a checking expression. At runtime, the checking expression corresponding to the executed control path is tracked using specialized hardware and the checking expressions corresponding to the control-path are executed. The checking expression recomputes the value of the critical variable and a mismatch between the recomputed and original values indicates an error. Experiments show that the derived detectors achieve low-overhead error detection (33%) while providing high coverage (77%) for errors that matter to the application (propagate and result in a crash). Future work will involve implementing the checking expressions derived in hardware (as part of the RSE [7]) and joint synthesis of the path-tracking module.

References

- [1] William R. Bush, Jonathan D. Pincus, and David J. Siela. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7), 2000
- [2] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. Programming Language Design and Implementation*, pages 57-68, June 2002.
- [3] D. Evans, J. Gutttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In *Proc. Symposium on the Foundations of Software Engineering (FSE)*, December 1994.
- [4] D. Engler, D.Y. Chen, S. Hallem, A. Chou., B. Chelf., Bugs as deviant behavior: A general approach to inferring errors in system code, *Proc. Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pp 57-72, 2001.
- [5] M. Hiller, Executable detectors for detecting data errors in embedded control systems, *Proc. International Conference on Dependable Systems and Networks (DSN)*, pp. 24-33, 2000.
- [6] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. In *Proc. International Conference on Dependable Systems and Networks (DSN)*, pages 135-144, 2002.
- [7] N. Nakka, J.Xu, Z.Kalbarczyk, R.K. Iyer., An architectural framework for providing reliability and security support, *Proc. Intl. Conference on Dependable Systems and Networks (DSN)*, pages: 585-594, 2004.
- [8] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1-25, 2001

- [9] Sudheendra Hangal and Monica Lam. *Tracking down software bugs using automatic anomaly detection*. In Proceedings of the International Conference on Software Engineering, May 2002
- [10] K.Pattabiraman, Z.Kalbarczyk, and R.K. Iyer, Application-based metrics for strategic placement of detectors Proc. 11th *International Symposium on Pacific Rim Dependable Computing (PRDC)*, pp. 75-82, December, 2005
- [11] K. Pattabiraman, G.P. Saggese, D. Chen, Z. Kalbarczyk, R.K. Iyer, "Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware," pp. 97-108, Sixth European Dependable Computing Conference (EDCC'06), 2006.
- [12] Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. Proceedings of the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'05), Sept 2005
- [13] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferro, *A C/C++ Source-to-Source Compiler for Dependable Applications*, Intl. Conference on Dependable Systems and Networks, 2000
- [14] N.S. Oh, S. Mitra and E.J. McCluskey, N. Oh, P. P. Shirvani, and E. J. McCluskey. *Error detection by duplicated instructions in super-scalar processors*. IEEE Transactions on Reliability, 51(1):63--75, March 2002.
- [15] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, SWIFT: Software Implemented Fault Tolerance, Proceedings of the 3rd International Symposium on Code Generation and Optimization
- [16] Nithin Nakka, "Processor-level error-detection and recovery", PhD Thesis, University of Illinois Urbana-Champaign, 2006.
- [17] N.S. Oh, S. Mitra and E.J. McCluskey, N. Oh, P. P. Shirvani, and E. J. McCluskey. *ED4I: Error Detection by diverse data and duplicated instructions in super-scalar processors*. IEEE Transactions on Reliability, 51(1):63-75, March 2002
- [18] Kim, M., S. Kannan, I. Lee, O. Sokolsky and M. Viswanathan, Java-mac: a run-time assurance tool for java programs, in: K. Havelurid and G. Rosu, editors, Proceedings of the First Workshop on Runtime Verification (RV'01), Paris, France, July 2001, Electronic Notes in Theoretical Computer Science 55 (2001).
- [19] Klaus Havelund and Grigore Rosu. Java PathExplorer - -- A runtime verification tool. In Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS'01, Montreal, Canada, June 18--22, 2001.
- [20] Mark Sullivan, Ram Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. In Proceedings of IEEE Twenty-Second Annual International Symposium on Fault-Tolerant Computing, July 8-10, 1992, Boston, Massachusetts.
- [21] Mark Weiser. Program slicing. In Proceedings of the 5th International Conference on Software Engineering, pages 439--449. IEEE Computer Society Press, 1981.
- [22] F. Tip. A survey of program slicing techniques. Journal of programming languages, 3:121--189, 1995.
- [23] C. Lattner and V. Adve. *LLVM: A compilation framework for lifelong program analysis & transformation*. In ACM Symp. on Code Generation and Optimization (CGO'04), Palo Alto, CA., 2004.
- [24] R. Cytron, J. Ferrante, B. Rosen, M. Wegman and F. Zadeck, Efficiently computing static single assignment form and the control dependence graph,, ACM Trans. on Programming Languages and Systems 13(4) 1991 pp.451-490.
- [25] Ranjit Jhala and Rupak Majumdar, Path Slicing, Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI), pp. 38-47, 2005.
- [26] I. Lee and R. K. Iyer, Software Dependability in the Tandem GUARDIAN System, IEEE Trans. on Software Engineering, Vol. 21, No. 5, pp. 455-467, May 1995.
- [27] W. Gu, Z. Kalbarczyk, R.K. Iyer, Z. Yang, Characterization of Linux Kernel Behavior under Errors, *Proc. International Conference on Dependable Systems and Networks (DSN'03)*, pp. 459-468, June 2003.
- [28] M. Carlisle and A. Rogers. Software caching and computation migration in Olden. In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Santa Barbara, CA, July 1995.
- [29] George A. Reis, David I. August, Robert Cohn, and Shubhendu S. Mukherjee, "Software Fault Detection Using Dynamic Instrumentation", Proceedings of the Fourth Annual Boston Area Architecture Workshop (BARC), February 2006.
- [30] Tao Zhang, Xiaotong Zhuang, Santosh Pande, Wenke Lee: Anomalous path detection with hardware support. Proceedings of Compilers, Architecture and Synthesis for Embedded Systems (CASES) pp. 43-54, 2005.
- [31] Kapil Vaswani Thazhuthaveetil, M.J. Srikant, Y.N, A programmable hardware path profiler, Proceedings of Code Generation and Optimization (CGO), pp. 217-228, 2005.
- [32] S. Chandra and P. M. Chen. How fail-stop are faulty programs ? In Proceedings of the 28th Symposium on Fault-Tolerant Computing, June 1998.
- [33] Jim Gray. Why do computers stop and what can be done about it? In Proc. Fifth Symposium on Reliability in Distributed Software and Database Systems, pages 3--12, 1986
- [34] http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_ii_pro_fpgas/product_table.htm:
- [35] S. Muchnick, "Advanced Compiler Design and Implementation", Morgan Kaufman, 1997.