Automated Derivation of Application-aware Error Detectors Using Static Analysis: The Trusted Illiac Approach

Karthik Pattabiraman, *Member IEEE,* Zbigniew Kalbarczyk, *Member IEEE,* and Ravishankar K. Iyer, *Fellow IEEE*

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 continuously 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.*

Index Terms—**B.2.3** Error Checking, **B.8.1** Reliability, Testing, and Fault-tolerance, **C.3.e** Reconfigurable Hardware, **D.2** Software Engineering (Reliability), **D.4.5.d** Fault-tolerance

1 INTRODUCTION

This paper presents a methodology to derive error detectors for an application based on compiler-based static analysis. The derived detectors detect data errors in the application. A data error is defined as a divergence in the data values used in a program from an error-free run of the program for the same input. 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 in timing and synchronization) [3], which are not caught by static or dynamic methods.

Furthermore, programs upon encountering an error, may execute for billions of cycles before crashing (if they crash) [4], during which time the error may propagate to a permanent state [5]. In order to detect runtime errors, we need mechanisms that can provide high-coverage, low-latency error detection to preempt uncontrolled system crash or hang and prevent error propagation that can lead to state corruption. This is the focus of this paper.

Duplication has traditionally been used to provide high coverage at runtime for software errors and hardware errors [6]. 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 techniques compare the results of replicated instructions at selected program points, such as stores to memory [7, 8]. 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 typically result in application failure (crash, hang, or incorrect output) [9]. Therefore, more than 50% of the errors detected by duplication are benign [10].

The main contribution of this paper is an approach to derive runtime error detectors using static analysis of the application. The derived detectors can be implemented using either software or programmable hardware. While this paper focuses on the software implementation of the detectors, the detectors have also been implemented in hardware in the context of the Reliability and Security Engine (RSE)[11]. They have been prototyped as part of the Trusted Illiac project, which is a configurable, application-aware, high-performance platform for trustworthy computing being developed at the University of Illinois [12, 13].

We find experimentally that the derived detectors preempt crashes and provide high detection coverage for errors that result in application failures. The key findings of the study are as follows: (1) the derived detectors detect around 75% of errors that propagate and cause crashes, (2) the percentage of benign errors detected is less than 3%, and (3) the average performance overhead of the derived detectors is 33%.

- *K. Pattabiraman is with Microsoft Research, Redmond, WA 98052. E-mail: karpat@microsoft.com. This work was performed when he was with the Center for Reliable and High-Performance Computing, Urbana, IL.*
- *Z. Kalbarczyk is with the Center for Reliable and High-Performance Computing, Urbana, IL 61801. E-mail: kalbarcz@uiuc.edu*
- *R.K. Iyer is with the Center for Reliable and High-Performance Computing, Urbana, IL 61801. E-mail: rkiyer@uiuc.edu*

2 Related Work

Related techniques for (1) uncovering software bugs using static/ dynamic program analysis and (2) providing runtime detection of hardware/ software errors can be divided into several broad groups as shown in Table 1.

The static techniques discussed in Table 1 are geared toward detecting errors at compile time, while the dynamic analysis techniques are geared towards providing feedback to the programmer for bug finding. Both these types are *fault-avoidance* techniques (the fault is removed before the program is operational) [14]. Despite the existence of these techniques and rigorous program testing, subtle but important errors such as timing errors persist in a program [3, 15]. Furthermore, 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 [10]. 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.

Table 1: Classification of related techniques

Class	Example	Comments
Static Analysis Techniques	Prefix [16], ESP [17], LINT[1]	Checks 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 finding errors that will never occur in a real execution, leading to wasteful detections.
Dynamic Analysis Techniques	DAIKON [2]	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. The invariants are derived based on the execution of the application with a representative set of inputs that are not in this set may result in the invariants being violated even when there is no error in the progam.
	DIDUCE [18]	Uses the invariants learned during an early stage of the program execution to detect errors in the subsequent part of the execution. It is unclear how the invariants learned during the early stages represent the entire application's execution. This in turn may lead to false detections.
Rule-based Detectors	Hiller et al. [19], Pattabiraman et al. [20]	Derives error detectors based on rule-based templates, wherein the choice of templates and the parameters are either manually specified [19] or automatically derived [20]. 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. Further, the rules learned may not be representative of all inputs to the application and may be violated even when there is no error in the application.
Inferring Specifications from Code	PR-Miner [21], Engler et al [22]	Learns program patterns from source code analysis and consider violations of these patterns as program bugs. 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. 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 code sections. Further, these techniques have false-positives i.e. many errors are not real bugs.
Compiler-based Replication	Benso et al.[23], EDDI [7], SWIFT [8]	Replicates the entire program, which can result in high performance overheads (90-100%). 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). This constitutes a wasteful detection (and subsequent recovery) from the application's viewpoint. Further, duplication-based techniques offer limited protection from software faults and permanent hardware faults because both the original program and the replica can incur common-mode faults.
Runtime Verification	JavaMac [24], JavaPathExplorer [25]	Checks whether the program violates a programmer-specified safety property 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 only consider errors that are directly detectable (by the checking technique), the coverage for a random hardware or software error is not clear.
Runtime Error Detection for specific error classes	Memory Safety Checking [26-28]	Checks every program store that is performed through a pointer (at runtime) to ensure that the write is within the allowed bounds of the pointer. The techniques are effective for detecting common problems due to buffer overflows and dangling pointer errors. It is unclear whether they are effective in detecting random errors that arise due to incorrect computation unless such an error results in a pointer writing outside its allowed bounds. The techniques also requires checking every memory write, and this can result in prohibitive performance overheads (5x-6x).
	Race Condition Detection [29, 30]	Checks for race conditions in a multi-threaded program. A race condition occurs when a shared variable is accessed without explicit and appropriate synchronization. The techniques check for races in lock-based programs by dynamically monitoring lock acquisitions and releases. However, these approaches involve instrumenting and dynamically monitoring memory writes to shared variables in programs, which in turn can result in prohibitve performance overheads (6x to 60x). Moreover, conventional race-detection techniques may find races that have no impact on the program's output (benign races), thereby resulting in wasteful detections.
	Control-flow Checking [31] [32]	Ensures that a program's statically derived control-flow graph is preserved during the program's execution. This is achieved by adding checks on the targets of jump instructions and at entries and exits of basic blocks. However, fault-injection experiments have shown that only 33% of the manifested errors result in violations of control-flow and can hence be detected by these techniques (even assuming that the detection coverage is 100%).

In earlier work [33], we have shown the feasibility of deriving error detectors based on static analysis of applications and have shown that the derived detectors provide high detection coverage (for data errors) with low performance overheads. This paper extends this idea by (1) presenting algorithms for automated static derivation of error detectors and their implementation, (2) discussing their scalability of the derivation process and the coverage of the derived detectors, and (3) qualitatively analyzing the coverage of the derived detectors for software errors.

3 APPROACH

This section presents an overview of the detector derivation approach.

3.1 Terms and Definitions

Backward program slice of a variable at a program location is the set of all program statements/ instructions that can affect the value of the variable at that program location [34].

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

Checking expression is an optimized sequence of instructions that recompute the critical variable. *It is computed from the backward slice of the critical variable for a specific acyclic control path in the program.*

Detector is the set of all checking expressions for a critical variable, one for each acyclic, intra-procedural control path in the program.

3.2 Steps in Detector Derivation

The main steps in error detector derivation are as follows:

A. Identification of critical variables. The critical variables are identified based on an analysis of the dynamic execution of the program. The application is executed with representative inputs to obtain its dynamic execution profile, which is used to choose critical variables for detector placement. Critical variables are variables with the highest dynamic fanouts in the program, as errors in these variables are likely to propagate to many locations in the program and cause program failure. The approach for identifying critical variables was presented in [35], where it was shown (experimentally) to provide 85% coverage with approximately 10 critical variables in the entire program[1]. However, in this paper, critical variables are chosen on a per-function basis in the program i.e. each function/ procedure in the program is considered separately to identify critical variables.

B. 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 (based on source language semantics). Hence, the slice will be a superset of the dependencies encountered during an execution of the program and encompasses all valid inputs.

C. 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.
- *Instrumentation:* Program is instrumented to track control-paths followed at runtime in order to choose the checking expression for that specific control path.

D. Runtime checking in hardware and software. The control path followed is tracked (by the inserted instrumentation) at runtime. The path-specific inserted checks are executed at appropriate points in the execution depending on the control path followed at runtime. 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.

The main sources of performance overhead for the detectors are as follows:

(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) *[11].* 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 component is implemented as a module in the RSE (Appendix A).

(2) *Checking:* In order to further reduce the performance overhead, the check execution itself can be moved to hardware. This is an area of future investigation.

3.3 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. Only the instructions in the backward slice of variable *f* are shown in Figure 1.

In Figure 1, 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 checking

[1] The paper considered ideal detectors which could detect any deviation from the correct value.

expression that checks the value of *f* along that path. 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 inserted check, the appropriate 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 program. In case there is a mismatch, an error is declared and the program is stopped.

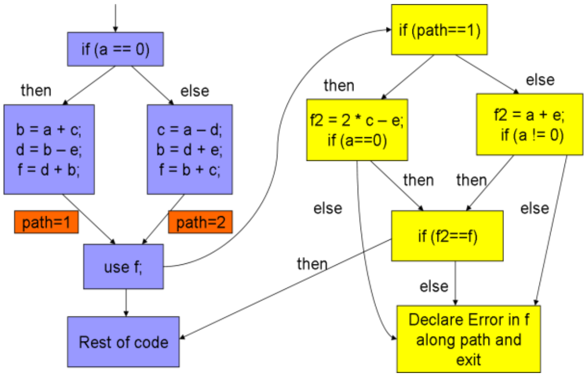


Figure 1: Example code fragment with detectors inserted

3.4 Software Errors Covered

Since the technique proposed in this paper enforces the compiler-extracted source-code semantics of programs at runtime, it can detect any software error that violates the source program's semantics at runtime. This includes software errors caused by pointer corruptions in programs (memory corruption errors) as well as those caused by missing or incorrect synchronization in concurrent programs (timing errors). We consider how the proposed technique detects these errors:

Memory corruption errors: Languages such as C and C++ allow pointers to write anywhere in memory (to the stack and heap). Memory corruption errors are caused by pointers in the code writing outside their intended object[2] (according to source code semantics), therby corrupting other objects in memory. However, static analysis performed by compilers typically assumes that objects are infinitely far apart in memory and that a pointer can only write within its intended object. As a result, the backward slice of critical variables extracted by the compiler includes only those dependences that arise due to explicit assignment of values to objects via pointers to the object. Therefore, the technique detects all memory errors that corrupt one or more variable in the backward slice of critical variables, as long as the shared state between the check and the main program is not corrupted (e.g. memory errors that affect function parameters will not be detected, as only intra-procedural slices are considered by the technique).

Figure 2 illustrates an example of a memory corruption error in an application and how the proposed technique detects the error. In the figure, function *foo* computes the running sum (stored in *sum)* of an array of integers (*buf)* and also the maximum integer (*max*) in the array. If the maximum exceeds a predetermined threshold, the function returns the accumulated sum corresponding to the index of the maximum element in the array (*maxIndex*).

In Figure 2, the array *sum* is declared to be of size *bufLen*, which is the number of elements in the array *buf.* However, there is a write to *buf[i+1]* in line 5, where *i* can take values from 0 to *bufLen*. As a result, a buffer overflow occurs in the last iteration of the loop, leading to the value of the variable *max* being overwritten by the write in line 5 (assuming that *max* is stored immediately after the array *buf* on the stack*)*. The value of *max* would be subsequently overwritten with the value of the sum of all the elements in the array, which is something the programmer almost certainly did not expect (this results in a logical error).

```
int foo(int buf[]) {
1:      int sum[bufLen];
2:      int max = 0; int maxIndex = 0;
3:      sum[0] = 0;
4:      for (int i = 0; i < bufLen; ++i) {
5:          sum[i + 1] = sum[i] + buf[i];
6:          if (max < buf[i])  {
7:                  max = buf[i];
8:                  maxIndex = i;
9:          }
10:     }
11:     if (max > threshold)     return sum[maxIndex];
12:     return sum[bufLen];
}
```

Figure 2: Example of a memory corruption error

In the above example, assume that the variable *max* has been identified as critical, and is being checked in line 9. Recall that the proposed technique will detect a memory corruption error *if and only if* the error causes corruption of the critical variable (which is the case in this example). In this case, the checking expression for *max* will depend on whether the branch corresponding to the *if* statement in line 6 is taken. If the branch is not taken, the value of *max* is the value of *max* from the previous iteration of the loop. If the branch is taken, then the value of *max* is computed to be the value *buf[i].* These are the only possible values for the *max* variable, and both values are represented in the detector. The memory corruption error in line 5 will overwrite the variable *max* with the value *sum[bufLen],* thereby causing a mismatch in the detector's value. Hence, the error will be detected by the technique.

Note that the detector does not isolate the actual line of code or the variable where the memory error occurs. Therefore, it can detect any memory corruption error that affects the value of the critical variable, independent of where the error occurs. As a result, the technique does not need to instrument all unsafe writes to memory as done by conventional memory-safety techniques (e.g. [26-28]).

[2] The term object refers to both program variables and memory objects.

Race conditions and synchronization errors: Race conditions occur in concurrent programs due to lack of synchronized accesses to shared variables. Static analysis techniques typically do not take into account asynchronous modifications of variables when extracting dependences in programs. As a result, the backward slice only includes modifications to the shared variables made under proper synchronization. Hence, race conditions that result in unsynchronized writes to shared variables to the variables in the backward slice of critical variables. However, race conditions that result in unsynchronized reads may not be detected unless the result read by the read propagates to the backward slice of the critical variable. Note that the technique does not detect benign races (i.e. race conditions in which the value of the variable is not affected by the order of the writes), as it checks the value of the variable being written to rather than whether the write is synchronized.

Figure 3 shows a hypothetical example of a race condition in a program. Function *foo* adds a constant value to each element of an array *a* which is passed into it as a formal parameter. It is also passed an array *a_lock,* which maintains fine-grained locks for each element of *A*. Before operating on an element of the array, the thread acquires the appropriate lock from the array *a_lock.* This ensures that another thread is not able to modify the contents of array *a[i], provided the other thread tries to acquire the lock before modifying a[i].* Therefore, the locks by themselves do not protect the contents of *a[i]* unless all threads adhere to the locking discipline. The property of adherence to the locking discipline is hard to verify using static analysis alone because (1) the thread modifying the contents of array *a* could be in a different module than the one being analyzed, and the source code of the other module may not be available at compile time, and (2) precise pointer analysis is required to find the specific element of *a* being written to in the array. Such precise analysis is often unscalable, and static analysis techniques perform approximations that result in missed detections.

```
1: void foo(int* a, mutex* alock, int n, int c) {
 2:      int i = 0;
 3:      int sum = 0;
 4:      for (i=0; i<n; i++) {
 5:           acquire_mutex( alock[i] );
 6:           old_a = a[i];
 7:            a[i] = a[i] + c;
 8:            check( a[i] == old_a + c)
 9:            release_mutex( alock[i] );
 10:     }
}
```

Figure 3: Example for race condition detection

The proposed technique, on the other hand, would detect illegal modifications to the array *a* even by threads that do not follow the locking discipline. Assume that the variable *a[i]* in line 7 has been determined to be a critical variable. The proposed technique would place a check on *a[i]* to recompute it in line 8. Now assume that the variable *a[i]* was modified by an errant thread that does not follow the locking discipline. This would cause the value of *a[i]* computed in line 7 to be different from what it should have been in a correct execution (which is its previous value added to the constant *c*). Therefore, the error is detected by the recomputation check in line 8.

The following points can be noted in the example: (1) The source code of the errant thread is not needed to derive the check and hence it can be in a different module, (2) The check will fail only if the actual computed value is different and is therefore immune to benign races that have no manifestation on the computation of the critical variable, and (3) in this example, it is enough for the technique to analyze the code of the function *foo* to derive the check for detecting the race condition[3].

3.5 Hardware Errors Covered

Hardware transient errors that result in corruption of architectural state are considered in the fault-model. Examples of hardware errors covered include,

- **Instruction fetch and decode errors:** Either the wrong instruction is fetched, (OR) a correct instruction is decoded incorrectly resulting in data value corruption.
- **Execute and memory unit errors:** An ALU instruction is executed incorrectly inside a functional unit, (OR) the wrong memory address is computed for a load/ store instruction, resulting in data value corruption.
- **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).

4 STATIC ANALYSIS

This section describes the static analysis technique to derive detectors and add instrumentation for path tracking. The bubble-sort program shown in Figure 4(a) is used as a working example throughout this section.

We use the LLVM compiler infrastructure [36]to derive error detectors for the program. A new compiler pass called the *Value Recomputation Pass (VRP)* was introduced into LLVM. 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.

LLVM uses Static Single Assignment form (SSA) [37] 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. This unique name makes it easy to

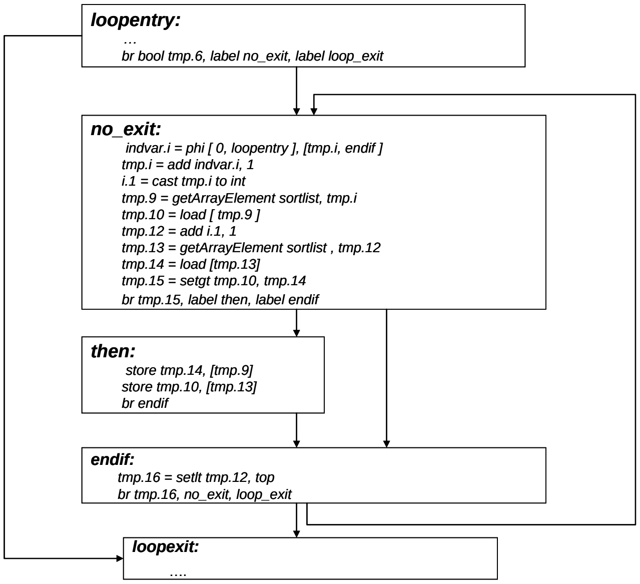
[3] This may not hold in case the modification is done prior to the function call.

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 specialization of the backward slice based on control-paths by the technique.

```
void Bubble(int srtElements, int* sortList) {
             int i, j,  top;
            bInitarr( sortList, srtElements );
            top=srtelements;
           while ( top>1 ) {//Outer-while-loop
                i=1;
               while ( i<top ) {// Inner while-loop
                     if ( sortlist[i] > sortlist[i+1] )
                    {
                           j = sortlist[i];
                          sortlist[i] = sortlist[i+1];
                          sortlist[i+1] = j;
                    } // end-if
                    i=i+1;
               } // end-inner-while
              top=top-1;
          } // end-outer-while
     }
```

(a)



```
loopentry:
    …
    br bool tmp.6, label no_exit, label loop_exit

no_exit:
     indvar.i = phi [ 0, loopentry ], [tmp.i, endif ]
    tmp.i = add indvar.i, 1
    i.1 = cast tmp.i to int
    tmp.9 = getArrayElement sortlist, tmp.i
    tmp.10 = load [ tmp.9 ]
    tmp.12 = add i.1, 1
    tmp.13 = getArrayElement sortlist , tmp.12
    tmp.14 = load [tmp.13]
    tmp.15 = setgt tmp.10, tmp.14
    br tmp.15, label then, label endif

then:
     store tmp.14, [tmp.9]
    store tmp.10, [tmp.13]
    br endif

endif:
   tmp.16 = setlt tmp.12, top
   br tmp.16, no_exit, loop_exit

loopexit:
      ....
```

(b)

Figure 4: (a) Example code fragment (b) Corresponding LLVM intermediate code

A simplified version of the LLVM intermediate code corresponding to the inner-while loop in the bubble-sort program is shown in Figure 4(b). In Figure 4(b), the basic blocks are labeled with unique names and their successors are shown through directed arrows. Each instruction assigns its result (if any) to a unique variable. The, *phi* instruction was explained earlier. The *getArrayElement* instruction dereferences an array base pointer and index to reference the element at the array index location. The *cast* instruction converts values of one type into another. The *setgt* and *setlt* instructions compare two values and the *br* instruction executes a branch based on the results of the comparison. The *load* and *store* instructions read from and write to memory respectively.

4.1 Value Recomputation Pass

The basic ideas behind the VRP were introduced by us in [33]. The details of the VRP algorithm are presented for the first time in this paper. The VRP takes LLVM intermediate code annotated with critical variables and extracts their path-specific backward slices. It computes the backward slice by traversing the static dependence graph of the program starting from the instruction that computes the value of the critical variable.

By extracting the path-specific backward slice and exposing it to other optimization passes in the compiler, the Value Recomputation Pass (VRP) enables aggressive compiler optimizations to be performed on the slice that would not be possible otherwise.

4.1.1 Path-specific Slicing Algorithm

An important contribution of this paper is the algorithm used for creating the path-specific slice for critical variables. The instruction that computes the critical variable in the program is called the critical instruction. In order to derive the backward program slice of a critical instruction, the VRP performs backward traversal of the static data dependence graph. The traversal starts from the critical instruction and terminates when one or more of the following conditions is met:

- **The beginning of the current function is reached**. 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 4a, the array *sortList* is passed as an argument to the function *Bubble*. The slice does not include the computation of *sortList* in the calling function. If *sortList* is a critical variable in the calling function, say *foo*, then a detector will be derived for it when *foo* is analyzed.
- **A basic block is revisited in a loop**. During the backward traversal, if data dependence within a loop is encountered, the detector is broken into two detectors, one placed on the critical variable and one on the variable that affects the critical variable within the loop. This second detector ensures that the variable within the loop is computed correctly and hence the value can be used without recomputing it in the first detector. Therefore, only acyclic paths are considered.
- **A dependence across loop iterations is encountered**. Recomputing critical variables across multiple loop iterations can involve loop unrolling or buffering intermediate values that are rewritten in the loop. This in turn can complicate the design of the detector. Instead, the VRP splits the detector into two detectors, one for the dependence-generating variable and one for the critical variable.
- **A memory operand is encountered**. Memory dependences are not considered because LLVM pro-

motes most memory objects to registers prior to running the VRP. Since there is an unbounded number of virtual registers for storing variables in SSA form, the analysis does not have to be constrained by the number of physical registers available on the target machine. However it may not always be possible to promote memory objects to register e.g. pointer references to dynamically allocated data. In such cases, the VRP duplicates the load of the memory object, provided the load address is not modified along the control path from the load instruction to the critical instruction (as determined by pointer analysis [38]).

Table 2: Pseudocode of backward traversal algorithm

```
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
```

The algorithm for computing path-specific backward slices of the critical instruction is shown in Table 2. We highlight its main points here:

- During the backward traversal, when a *phi*-instruction is encountered indicating a merge in control-flow paths, the slice is forked for each control path that is merged at the *phi*. The algorithm maintains the list of instructions in each path-specific slice in the array *SliceList*. The function *computeSlices* takes as input the critical instruction and outputs the *SliceList* array, which contains the instructions in the backward slice for each acyclic path in the function.
- 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 *computeSlices* function calls the *visit* function for each critical instruction. The *visit* function visits each operand of an instruction in turn, adding it to the *SliceList* of the current path. When a *phi* instruction is encountered, a new path is spawned 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.
- Certain instructions cannot be recomputed in the checking expression, because performing recomputation of such instructions can alter the semantics of the program. Examples are *malloc*s, *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 (section 3.2).

Assuming that the critical variable chosen for the example in Figure 4a is *sortlist[i]*, the intermediate code representation for this variable is the instruction *tmp.10* in Figure 4b. The VRP computes the backward slice of *tmp.10*, which consists of the two paths shown in Figure 5.

Path 0: no_exit → loopentry	***Path 1: endif → loopentry***
indvar.i = 0 *tmp.i = add indvar.i, 1* *tmp.9 =getArrayElement sortlist,tmp.i* *tmp.10 = load[tmp.9]*	*indvar.i = tmp.i* *tmp.i = add indvar.i, 1* *tmp.9 = getArrayElement sortlist,tmp.i* *tmp.10 = load [tmp.9]*

Figure 5: Path-specific slices for example

4.1.2 VRP and Other Optimization Passes

After extracting the path-specific slices, the VRP performs the following operations on the slices:

- Places the instructions in the backward slice of the critical variable corresponding to each control path in its own basic block.

- Replaces the *phi* instructions in the slice with the incoming value corresponding to the control edges for the path. This allows subsequent compiler optimization passes to substitute the *phi* values directly in their uses through either constant propagation or copy propagation [38].
- Creates copies of variables used in the path-specific slices that are not live at the detector insertion point. For example, the value of *tmp.i* is overwritten in the loop before the detector can be reached and a copy *old.tmp.i* is created before the value is overwritten.
- Renames the operands in the slices to avoid conflicts with the main program and thereby ensure that SSA form is preserved by the slice.
- Instruments program branches with path identifiers considered by the backward slicing algorithm. This includes introduction of special instructions at branches pertaining to the paths in the slice, and also at function entry and exit points.

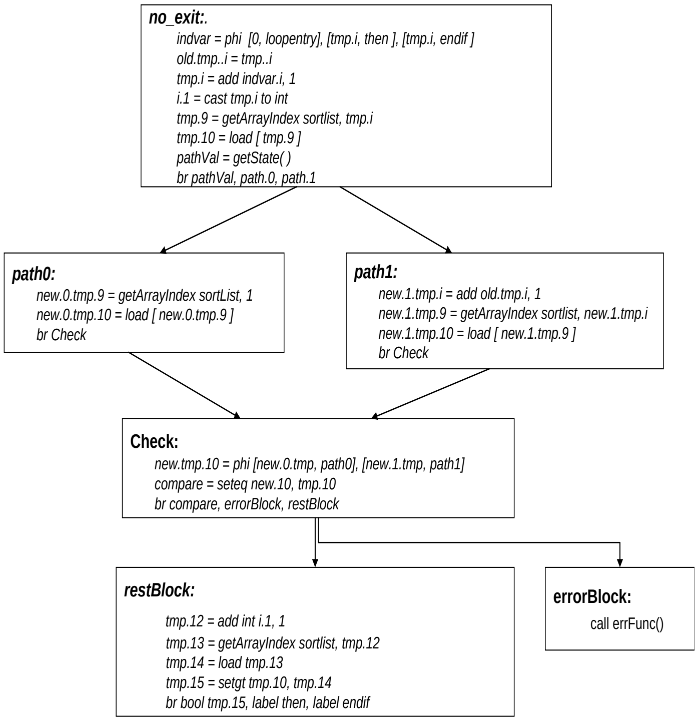
The standard LLVM optimization passes are invoked on the path-specific backward slices extracted by the VRP. The optimization passes yield reduced instruction sequences that compute the critical variables for the corresponding paths. Further, since there are no control-transfers within the sequence of instructions for each path, the compiler is able to optimize the instruction sequence for the path much more aggressively than it would have otherwise. This is because the compiler does not usually consider specific control paths when performing optimizations for reasons of space and time efficiency. *However, by selectively extracting the backward slices for critical variables and by specializing them for specific control paths, the VRP is able to keep the space and time overheads small.*

4.1.3 VRP Output

The LLVM intermediate code from Figure 4 with the checks inserted by the VRP is shown in Figure 6. 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 6. The optimized set of instructions corresponding to the first control path is encoded as a checking expression in the block *path0* in Figure 6. The second control path corresponds to the control transfer from the basic block *endif* to the basic block *no_exit* in Figure 4. The optimized set of instructions corresponding to the second control path is encoded as a checking expression in the block *path1* in Figure 6.

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 and comprise of 2 instructions and 3 instructions respectively. The basic block *Check* in Figure 6 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* in the original program.

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 4(b) (this instruction indirectly impacts the computation of the critical variable *tmp.10).*



```
no_exit:.
    indvar = phi  [0, loopentry], [tmp.i, then ], [tmp.i, endif ]
    old.tmp..i = tmp..i
    tmp.i = add indvar.i, 1
    i.1 = cast tmp.i to int
    tmp.9 = getArrayIndex sortlist, tmp.i
    tmp.10 = load [ tmp.9 ]
    pathVal = getState( )
    br pathVal, path.0, path.1
```

```
path0:
    new.0.tmp.9 = getArrayIndex sortList, 1
    new.0.tmp.10 = load [ new.0.tmp.9 ]
    br Check
```

```
path1:
    new.1.tmp.i = add old.tmp.i, 1
    new.1.tmp.9 = getArrayIndex sortlist, new.1.tmp.i
    new.1.tmp.10 = load [ new.1.tmp.9 ]
    br Check
```

```
Check:
    new.tmp.10 = phi [new.0.tmp, path0], [new.1.tmp, path1]
    compare = seteq new.10, tmp.10
    br compare, errorBlock, restBlock
```

```
restBlock:

    tmp.12 = add int i.1, 1
    tmp.13 = getArrayIndex sortlist, tmp.12
    tmp.14 = load tmp.13
    tmp.15 = setgt tmp.10, tmp.14
    br bool tmp.15, label then, label endif
```

```
errorBlock:
    call errFunc()
```

Figure 6: LLVM code with checks inserted by VRP

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.* In *path0,* the compiler performs constant propagation and replaces the computation of *tmp.i* with the constant *1* in Figure 6. 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. Thus, the error in the computation of *tmp.i* is detected.

Next, consider the case when the path followed corresponds to the execution of the checking expression in *path1*. The VRP 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. Thus, the error in the computation of *tmp.i* is detected.

4.2 Scalability

This section discusses factors that could potentially limit the scalability of the VRP algorithm and how these are addressed by the proposed technique. The factors that affect the technique's scalability are as follows:

- **Number of control paths:** This is addressed by considering only intra-procedural, acyclic paths in the program corresponding to the backward slices of critical variables. At worst, the number of paths is exponential in the number of branch instructions in the program. In practice however, the number of control paths is polynomial in the number of branch instructions (unless the program is performing decision tree like computations).

- **Size of checking expression:** The size of the checking expression depends on the number of levels in the dependence tree of the critical variable considered by the algorithm. Terminating the dependency tree at loop and function boundaries naturally limits the checking expression's size.

- **Number of detectors:** The number of critical variables per function is a tradeoff between the desired coverage and an acceptable performance overhead. Placing more detectors achieves higher coverage but may result in higher performance overheads. The algorithm may introduce additional detectors, for example, when splitting a detector into two detectors across loop iterations, but this reduces the size of each checking expression. Therefore, for a given number of critical variables, the number of detectors varies inversely as the size of each checking expression.

4.3 Coverage

The VRP operates on program variables at the compiler's intermediate representation (IR) level. In the LLVM infrastructure, the IR is close to the program's source code [36]and abstracts many of the low-level details of the underlying architecture. For example, the IR has an infinite number of virtual registers, uses Static Single Assignment (SSA), and has native support for memory allocation (*malloc* and *alloca*) and pointer arithmetic (*getElementPtr*[4] instruction). Moreover, the runtime mechanisms for stack manipulations and function calls are transparent to the IR. As a result, the VRP may not protect data that is not visible at the IR level. Therefore, the VRP is best suited for detecting errors that impact program state visible at the source level. Note that the generic approach presented in Section 3, however, is not tied to a specific level of compilation and can be implemented at any level.

The VRP operates on LLVM's intermediate code, which does not include common runtime mechanisms such as manipulation of the stack and base pointers. Moreover, the intermediate code assumes that the target machine has an infinite register file and does not take into account the physical limitations of the machine.

Data errors in a program can occur in three possible places (locations): (1) Source-level variables or memory objects, (2) Precompiled Libraries linked with the application, and (3) Code added by the compiler's target-specific code generator for common runtime operations such as stack manipulation and handling register-file spills. The technique presented in the paper aims at detecting errors in the first category, and can be extended to detect errors in the second category provided the source code of the library is available or the library is compiled with the proposed technique. However, errors in the third category, namely those that occur in the code added by the compiler's code generator cannot be detected using the proposed technique unless the error affects one or more source-level variables or memory objects. This is because the code added by the compiler is transparent to the VRP and hence cannot be protected by the derived detectors.

The steps in compiling a program with LLVM are as follows: First, the application's source code along with the source (or intermediate) code of runtime libraries are converted to LLVM's generic intermediate code form. This intermediate form is in-turn compiled onto the target architecture's object code, which is then linked with precompiled libraries to form the final executable. The process is similar to conventional compilation, except that the application and the source libraries are first compiled to the intermediate code format (by a modified gcc frontend) before being converted to object code. Each level of compilation progressively adds more state to the program. Table 3 shows the data elements of the program's state visible at each level of compilation.

As shown in Table 3, the intermediate code level does not include data elements in the final executable that are added by the compiler and linker. Since the VRP operates at the intermediate code level, it does not see the elements in the lower levels and the derived detectors may not detect errors in these levels. This can be addressed by implementing the technique at lower compilation levels.

Table 3: State visible at each level of compilation

Code Level	Elements of program state that are visible
Source Level	(1) Local variables, (2) global variables and (3) dynamic data allocated on heap
Intermediate Code	(1) Branch addresses of *if* statements, loops, and case statements, (2) temporary variables used in evaluation of complex expressions
Object Code	(1) Temporary variables to handle register file spills, and (2) stack manipulation mechanisms.

4.4 State Machine Generation

The VRP extracts a set of checking expressions for each detector in the program. Each checking expression in the set corresponds to an acyclic, intra-procedural control path leading to the critical variable from the top of the

[4] This is the general case of the *getArrayElement* instruction.

function. The VRP also inserts instrumentation to notify the runtime system when the program takes a branch belonging to one of the paths in the set. This is done by inserting a special operation called *EmitEdge* that identifies the source and destination basic blocks of the branch with unique identifiers. The VRP then exports the basic block identifiers of the branches along each path in a separate text file for each detector in the program.

A post-processing analysis then parses these text files and builds a state-machine representation of the paths for each check. The state machines are constructed such that every instrumented branch in the program causes state transitions in one or more state machines. A complete sequence of branches corresponding to a control path for which a checking expression has been derived will drive the state machine for the check to an accepting state corresponding to the checking expression.

Table 4: Algorithm to convert paths to state machines

```
for each critical variable V in the program:
        open the path-file corresponding to the variable
        for each path in the path-file:
              PathNumber ← Read path ID in path file
               Read an edge e = (src, sink) from the path file
               S ← Start_State
              Create an accepting state "A" for the path
              if this is the only edge for the path:
                   if Transition[S, A] does not contain e
                       Transition[S, A] <- Transition[S,A] U e
               else:
                   current = S
                   for each edge e in the path
                        if there exists a state K such that
                          (Transition[current,K] contains e):
                            current ← K
                        else:
                            Create a new state L
                            Transition[current, L] ← e
                            current ← L
                   Set current as the accepting state for path
          close the path file for the critical variable
endfor
```

The algorithm used by the post-processing analysis to convert the control edge sequences to finite state machines is shown in Table 4. The algorithm processes the path files for each check, and adds states to the state machine corresponding to the check. The aim is to distinguish one path from another in the check, while at the same time introducing the least number of states to the state machine. This is because each state occupies a fixed number of bits in hardware, and our goal is to minimize the total number of bits that must be stored by the hardware module for path-tracking and consequently the area occupied by it (see Appendix A).

The algorithm in Table 4 works as follows: It starts in the starting state of the state machine and processes each edge in the list of edges for the path. It adds a new state for an edge if and only if no transition exists for the edge from the current state in the state machine. If such a transition exists, it transitions to the state leading from the current state corresponding to the edge, and processes the next edge in the path. It continues until it has processed all the edges of the path, and marks the last state added as the accepting state for the path in the state machine. When the algorithm terminates, it outputs the transition table for the state machines, as well as the list of accepting states corresponding to each path of the check.

The time-complexity of the algorithm in Table 4 is *O(| V| * | P| * | E|)*, where | V| is the number of critical variables in the program, | P| is the maximum number of control-paths in the backward slice of the variable and | E| is the maximum number of edges in the control paths corresponding to each critical variable. The space complexity of the technique is *O (| V| *| Ů E| *H)*, where | H| is the maximum number of shared edges among control-paths corresponding to the critical variables, and Ů E is the union of the edges in the program's control paths.

Figure 7a shows the control-flow graph (CFG) of the program shown in Figure 4. As shown earlier, the critical variable is computed in the basic block *endif*. The VRP has identified 4 intra-procedural acyclic paths[5] in the backward slice of the critical variable:

1. loopentry → no_exit, no_exit→ endif
2. loopentry → no_exit, then → endif
3. endif → no_exit, no_exit → endif
4. endif → no_exit, then → endif

The state machine derived by the algorithm for the control-flow graph in Figure 7(a) is shown in Figure 7(b). The algorithm has introduced two new states *A* and *B* in addition to four accepting states *D, E, F* and *G* for the four paths shown above. The transitions between states correspond to the edges identified by the VRP to distinguish paths from one another.

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.

5.1 Performance Measurement

All experiments are carried out on a single core Pentium 4 machine with 1GB RAM and 2.0 Ghz clock speed running the Linux operating system. The performance overheads of each individual component introduced by the proposed technique are measured as follows:

Modification overhead: Performance overhead due to the extra code introduced by the *VRP* for instrumentation and checking. This code may cause cache misses and branch mispredictions and incur performance overhead.

[5] In the earlier discussion, only two of these paths were considered.

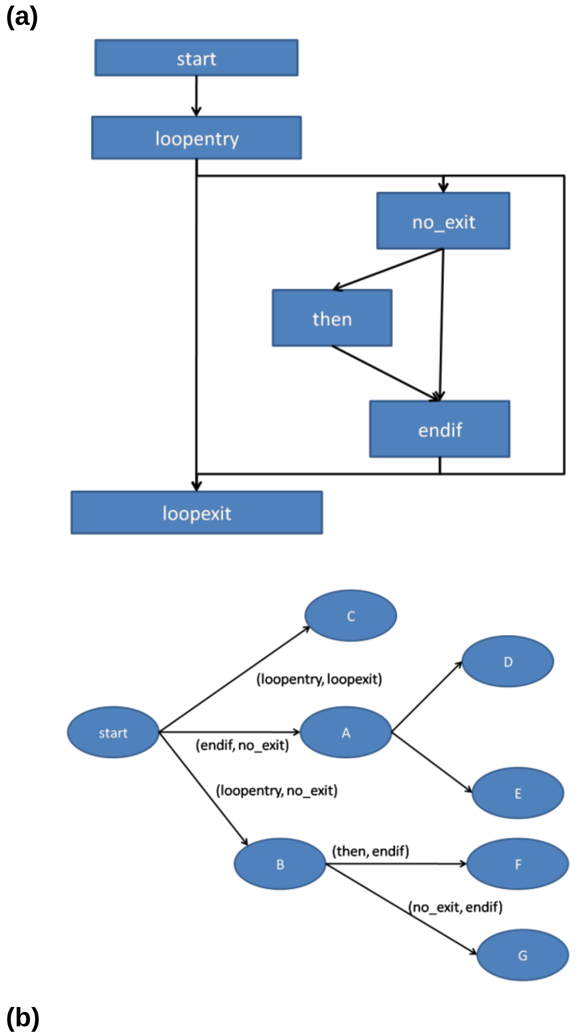


Figure 7(a) Control-flow graph (CFG) of bubblesort application (b) Corresponding finite state machine

Checking overhead: Performance overhead of executing the instructions in each check to recompute the critical variable and compare the recomputed value with the original value. This also includes the cost of branching to the check, choosing the checking expression to be excuted and branching back to the program's code.

The overhead of path-tracking is not considered in measuring performance overheads because the path tracking is done in parallel with the execution of the main program using a specialized hardware module. The path-tracking module can execute asynchronously and needs to be synchronized with the main processor only when the check is performed (see Appendix A for a detailed description).

We implemented the path-tracking module using software emulation and measured the performance overheads of the application with both path-tracking and checking enabled. We then measure the application overhead with only path-tracking enabled and subtract it from the earlier result in order to obtain the checking overheads. In order to obtain the code modification overheads, we executed the application with both path-tracking and checking disabled and measured the increase in execution time over the unmodified application.

Finally, we do not assume a specific recovery technique in the paper and hence do not consider the overhead of error recovery in our measurements.

5.2 Coverage Measurements

Fault injections: In order to measure the coverage of the derived detectors, we inject faults into the data of the application protected with the derived detectors. We implemented a new LLVM pass to insert calls to a special *faultInject* function after the computation of each program variable in the original program. The variable to be injected is 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 inserted for the variable.

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 (that are visible at the compiler's intermediate code level). In order to ensure controllability, only a single fault is injected in each execution of the application.

Only the values in the original function prior to instrumentation are considered for fault-injection. No faults are injected into the detectors themselves. This is because we assume that no more than one fault can occur during the application's execution. Injecting faults into detectors will at worst lead to false detections, i.e., detection of an error when none exists. However, we do inject errors into states shared between the detectors and the program in order to emulate common mode errors.

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 under the error can be observed. The coverage of the detector is classified based on the final outcome of the program. For example, a detector is considered to detect a crash if the detector upon encountering the error, flags a violation, and subsequently the program crashes. Hence, when a detector detects a crash, it is in reality, preempting the crash of the program.

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 (for example, the state can be recovered from a clean checkpoint). Hence, the 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 is executed after the fault has been injected. If the original value into which the error was injected is overwritten, the propagation of the error is no longer tracked. The error-propagation is tracked using instrumentation inserted into the program through a new LLVM pass. The instrumentation is inserted just before the definitions of variables that are dependent on the fault-injected value.

5.3 Benchmarks

Table 5 describes the programs used to evaluate the technique and their characteristics. The first 9 programs in the table are from the Stanford benchmark suite [39] and the next 5 programs are from the Olden benchmark suite [40]. The former benchmark set consists of small programs performing a multitude of common tasks. The latter benchmark set consists of pointer-intensive programs.

Table 5: Benchmark programs and characteristics

Bench mark	Lines of C	Description of program
IntMM	159	Matrix multiplication of integers
RealMM	161	Matrix multiplication of floating-point numbers
Oscar	270	Computes Fast-Fourier Transform
Bubblesort	171	Sorts a list of numbers using bubblesort
Quicksort	174	Sorts a list of numbers using quicksort
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 linked lists)
Em3d	639	Electro-magnetic wave propagation (linked lists)
Mst	389	Computes minimum spanning tree (graphs)
Barnes-Hut	1427	Solves N-body force computation problem(octrees)
Tsp	572	Solves traveling salesman problem (binary trees)

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 placement analysis.

6.1 Performance Overheads

The performance overhead of the derived detectors relative to the normal (uninstrumented) program's execution is shown in Figure 8. Both the checking overhead and the code modification overheads are represented. The results are summarized below:

- The average checking overhead introduced by the detectors is 25%, while the average code modification overhead is 8%. Therefore, the total performance overhead introduced by the detectors is 33%.
- The worst-case overheads are incurred in the case of the *tsp* application, which has a total overhead of nearly 80%. This is because *tsp* is a compute-intensive program involving tight loops. Checks within a loop introduce extra branch instructions and increase the execution time.

6.2 Detection Coverage

For each application, 1000 faults are injected, one in each execution of the application. The error-detection coverage (when 5 critical variables are chosen in each function) for different classes of failure are reported in Table 6. 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 fault injection experiments. The second column of the table shows the number of errors that propagate and lead to the application crashing. The numbers within the braces in this column indicate the percentage of propagated, crash-causing errors that are detected before propagation.

Table 6: Coverage with 5 critical variables / function

Apps	Prop. Crashes (%)	FSV (%)	Hang (%)	Success (%)
IntMM	100 *(97)*	100		9
RealMM	100 *(98)*			0
Oscar	57 *(34)*	7	60	0.5
Bubblesort	100 *(73)*	100	0	5
Quicksort	90 (57)	44	100	4
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**

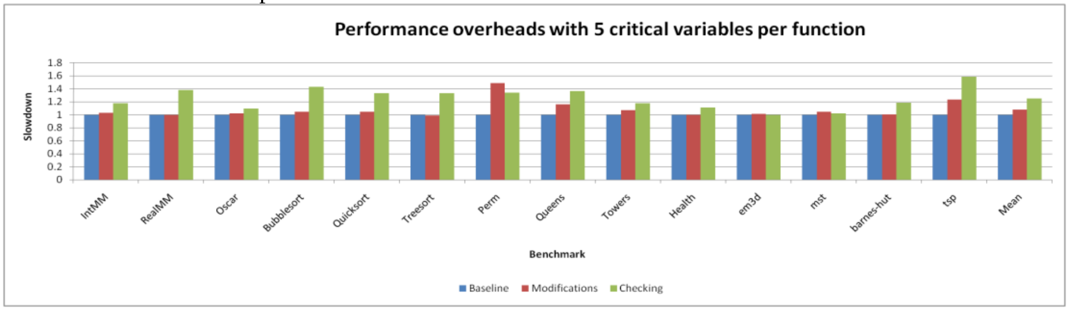


Figure 8: Performance overhead when 5 critical variables are chosen per function

6.3 Discussion

The results indicate the proposed technique achieves 77% coverage for errors that propagate and cause the program to crash. Full-duplication approaches can provide 100% coverage if they perform comparisons after every instruction. In practice, this is very expensive, and full-duplication approaches compare instructions only before store and branch instructions [7, 8]. With this optimization, the coverage provided by full-duplication is less than 100%. The papers that describe these techniques do not quantify the coverage in terms of error propagation, so a direct comparison with our technique is not possible. In an earlier study, we found that about 15% of the errors detected by full-duplication techniques resulted in a crash in the same cycle as the detection [10]. These detections are in effect redundant, as the error does not propagate prior to the crash. Therefore when excluding redundant detections, the proposed technique detects 90% of the errors detected by full-duplication. 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 [7, 8]. An important aspect of the technique is that it detects just 2.5% of benign errors in an application. In contrast, in full-duplication, over 50% of the detected errors are benign [9, 10].

7 Conclusion

This paper presented a technique to derive error detectors for protecting an application from data errors (due to both 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 based on specific control-paths in the application, to form a checking expression. At runtime, the control path executed by the progrm is tracked using specialized hardware, and the corresponding checking expressions are executed. The checking expressions recompute the values of the critical variable and check whether the recomputed value diverges from the original value computed in the program, in which case the program is halted.

Experiments show that the derived detectors achieve low-overhead error detection (33%) while providing high coverage (77%) for errors that cause application failure. Further, they detect less than 3 % of benign errors.

Future work will focus on (1) deriving detectors at lower levels of compilation (e.g. assembly code) in order to improve the detection coverage and (2) migration of the checking functionality to reconfigurable hardware in order to reduce the performance overheads of the detectors.

Acknowledgments: This work was supported in part by National Science Foundation (NSF) grants CNS-0406351, CNS-0524695, and CNS-05-51665, the Gigascale Systems Research Center (GSRC/ MARCO), Motorola Corporation as part of the Motorola Center for Communications (UIUC), and Boeing Corporation as part of Boeing Trusted Software Center at the Information Trust Institute. We thank Fran Baker for editorial support.

References

[1] Evans, D., J. Guttag, J. Horning, and Y.-M. Tan. *LCLint: a tool for using specifications to check code*. in *2nd ACM SIGSOFT symposium on Foundations of software engineering*. 1994. New Orleans, Louisiana, United States: ACM Press.

[2] Ernst, M.D., J. Cockrell, W.G. Griswold, and D. Notkin. *Dynamically discovering likely program invariants to support program evolution*. in *21st international conference on Software engineering*. 1999. Los Angeles, California, United States: IEEE Computer Society Press.

[3] Gray, J. *Why do computers stop and what can be done about it*. in *Symposium on Reliable Distributed Systems*. 1986: IEEE.

[4] Gu, W., Z. Kalbarczyk, R. Iyer, and Z. Yang. *Characterization of linux kernel behavior under errors*. in *International Conference on Dependable Systems and Networks*. 2003: IEEE Computer Society.

[5] Chandra, S. and P.M. Chen. *How Fail-Stop are Faulty Programs?* in *Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*. 1998: IEEE Computer Society.

[6] Spainhower, L. and W. Bartlett, *Commercial Fault Tolerance: A Tale of Two Systems.* IEEE Transactions on Dependable and Secure Systems, 2004. **1**(1): p. 87-96.

[7] Oh, N., P.P. Shirvani, and E.J. McCluskey, *Error detection by duplicated instructions in super-scalar processors.* IEEE Transactions on Relibility, 2002. **51**(1): p. 63-75.

[8] Reis, G.A., J. Chang, N. Vachharajani, R. Rangan, and D.I. August. *SWIFT: Software Implemented Fault Tolerance*. in *International symposium on Code generation and optimization*. 2005: IEEE Computer Society.

[9] Iyer, R.K., N.M. Nakka, Z.T. Kalbarczyk, and S. Mitra, *Recent advances and new avenues in hardware-level reliability support.* Micro, IEEE, 2005. **25**(6): p. 18-29.

[10] Nakka, N., K. Pattabiraman, and R. Iyer, *Processor-Level Selective Replication*, in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2007, IEEE Computer Society.

[11] Nakka, N., Z. Kalbarczyk, R.K. Iyer, and J. Xu. *An Architectural Framework for Providing Reliability and Security Support*. in *International Conference on Dependable Systems and Networks*. 2004: IEEE Computer Society.

[12] Iyer, R.K., Z. Kalbarczyk, K. Pattabiraman, W. Healey, W.-M.W. Hwu, P. Klemperer, and R. Farivar, *Toward Application-Aware Security and Reliability*. IEEE Security and Privacy, 2007. **5**(1): p. 57-62.

[13] Iyer, R.K. *TRUSTED ILLIAC: A Configurable Hardware Framework for a Trusted Computing Base*. 2007.

[14] Avizienis, A., J.C. Laprie, B. Randell, and C. Landwehr, *Basic concepts and taxonomy of dependable and secure computing*. IEEE Transactions on Dependable and Secure Computing, 2004. **1**(1): p. 11-33.

[15] Sullivan, M. and R. Chillarege. *Software defects and their impact on system availability-a study of field failures in operating systems*. in *Twenty-First Symposium on Fault-Tolerant Computing*. 1991.

[16] Bush, W.R., J.D. Pincus, and D.J. Sielaff, *A static analyzer for finding dynamic programming errors.* Software Practice and Experience, 2000. **30**(7): p. 775-802.

[17] Das, M., S. Lerner, and M. Seigle. *ESP: path-sensitive program verification in polynomial time*. in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 2002. Berlin, Germany: ACM Press.

[18] Hangal, S. and M.S. Lam. *Tracking down software bugs using automatic anomaly detection*. in *24th International Conference on Software Engineering*. 2002. Orlando, Florida: ACM Press.

[19] Hiller, M. *Executable Assertions for Detecting Data Errors in Embedded Control Systems*. in *International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*. 2000: IEEE Computer Society.

[20] Pattabiraman, K., G.P. Saggese, D. Chen, Z. Kalbarczyk, and R.K. Iyer. *Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware*. in *Sixth European Dependable Computing Conference*. 2006. Coimbra, Portugal: IEEE CS Press.

[21] Li, Z. and Y. Zhou. *PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code*. in *13th ACM SIGSOFT international symposium on Foundations of software engineering*. 2005. Lisbon, Portugal: ACM Press.

[22] Engler, D., D.Y. Chen, S. Hallem, A. Chou, and B. Chelf. *Bugs as deviant behavior: a general approach to inferring errors in systems code*. in *Eighteenth ACM Symposium on Operating systems principles*. 2001. Banff, Alberta, Canada: ACM Press.

[23] Benso, A., S. Chiusano, P. Prinetto, and L. Tagliaferri. *A C/C++ Source-to-Source Compiler for Dependable Applications*. in *International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*. 2000: IEEE Computer Society.

[24] Kim, M., M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, *Java-MaC: A Run-Time Assurance Approach for Java Programs*. Formal Methods in System Design, 2004. **24**(2): p. 129-155.

[25] Havelund, K. and G. Rosu, *An Overview of the Runtime Verification Tool Java PathExplorer*. Formal Methods in System Design, 2004. **24**(2): p. 189-215.

[26] Dhurjati, D., S. Kowshik, and V. Adve. *SAFECode: enforcing alias analysis for weakly typed languages*. in *ACM SIGPLAN conference on Programming language design and implementation*. 2006. Ottawa, Ontario, Canada: ACM Press.

[27] Jones, R.W.M. and P.H.J. Kelly. *Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs*. in *Automated and Algorithmic Debugging*. 1997.

[28] Ruwase, O. and M.S. Lam. *A practical dynamic buffer overflow detector*. in *11th Annual Network and Distributed System Security*. 2004.

[29] Savage, S., M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, *Eraser: a dynamic data race detector for multithreaded programs*. ACM Transactions on Computer Systems, 1997. **15**(4): p. 391-411.

[30] Engler, D. and K. Ashcraft, *RacerX: effective, static detection of race conditions and deadlocks*. SIGOPS Oper. Syst. Rev., 2003. **37**(5): p. 237-252.

[31] Oh, N., P.P. Shirvani, and E.J. McCluskey, *Control-flow checking by software signatures*. IEEE Transactions on Reliability, 2002. **51**(1): p. 111-122.

[32] Abadi, M., M. Budiu, U.l. Erlingsson, and J. Ligatti. *Control-flow integrity*. in *12th ACM conference on Computer and communications security*. 2005. Alexandria, VA, USA: ACM Press.

[33] Pattabiraman, K., Z. Kalbarczyk, and R. Iyer. *Automated Derivation of Application-Aware Error Detectors using Static Analysis*. in *Internation Online Testing Symposium (IOLTS)*. 2007: IEEE.

[34] Tip, F., *A survey of program slicing techniques*. Journal of Programming Languages, 1995. **3**(3): p. 121-189.

[35] Pattabiraman, K., Z. Kalbarczyk, and R.K. Iyer. *Application-based metrics for strategic placement of detectors*. in *Pacific Rim Dependable Computing*. 2005. Changsha, China: IEEE CS Press.

[36] Lattner, C. and V. Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis \ & Transformation*. in *international symposium on Code generation and optimization*. 2004. Palo Alto, California: IEEE Computer Society.

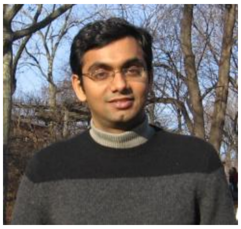
[37] Cytron, R., J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*. ACM Transactions on Programming Languages and Systems, 1991. **13**(4): p. 451-490.

[38] Muchnick, S.S., *Advanced compiler design and implementation*. 1997: Morgan Kaufmann Publishers Inc. 856.

[39] Weicker, R.P., *An Overview of Common Benchmarks*, in *Computer*. 1990. p. 65-75.

[40] Carlisle, M.C. and A. Rogers. *Software caching and computation migration in Olden*. in *Fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 1995. Santa Barbara, California, United States: ACM Press.

AUTHORS' BIOGRAPHY



Karthik Pattabiraman received the M.S and PhD degree in computer science from the University of Illinois at Urbana-Champaign (UIUC). He is currently a post-doctoral researcher at Microsoft Research. His research interests include design of reliable and secure applications using static and dynamic analysis, as well as experimental and formal techniques for dependability validation. Karthik's dissertation proposed the idea of application-aware dependability and he was the lead graduate student in the Trusted Illiac project at the University of Illinois. Based on his dissertation work, Karthik Pattabiraman was awarded the *William C. Carter* award in 2008 by the IFIP Working Group on Dependability (WG 10.4) and the IEEE Technical Committee on Fault-tolerant Computting (TC-FTC). He is a member of the IEEE and the IEEE Computer Society.

Zbigniew T. Kalbarczyk received the PhD degree in computer science from the Technical University of Sofia, Bulgaria. He is currently a principal research scientist at the Center for Reliable and High-Performance Computing in the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. After receiving his doctorate, he worked as an assistant professor in the Laboratory for Dependable Computing at Chalmers University of Technology in Gothenburg, Sweden. His research interests are in the area of reliable and secure networked systems. Currently, he is a lead researcher on the project to explore and develop high availability and security infrastructure capable of managing redundant resources across interconnected nodes, to foil security threats, detect errors in both the user applications and the infrastructure components, and recover quickly from failures when they occur. His research involves also developing of automated techniques for validation and benchmarking of dependable computing systems. He served as a program Chair of Dependable Computing and Communication Symposium (DCCS), a track of the International Conference on Dependable Systems and Networks (DSN) 2007 and Program Co-Chair of Performance and Dependability Symposium, a track of the DSN 2002. He is a member of the IEEE and IEEE Computer Society.

Ravishankar K. Iyer is Interim Vice Chancellor for Research at the University of Illinois at Urbana-Champaign, where he is a George and Ann Fisher Distinguished Professor of Engineering. He holds appointments in the Department of Electrical and Computer Engineering and the Department of Computer Science and his previous post was the Director of the Coordinated Science Laboratory (CSL) at Illinois. Professor Iyer also serves as Co-Director of the Center for Reliable and High-Performance Computing at CSL and Chief Scientist at the Information Trust Institute.Iyer's research interests are in the area of dependable and secure systems. He has been responsible for major advances in the design and validation of dependable computing systems. He currently leads the TRUSTED ILLIAC project at Illinois, which is developing application-aware adaptive architectures for supporting a wide range of dependability and security requirements in heterogeneous environments. Professor Iyer is a Fellow the AAAS, the IEEE and the ACM. He has received several awards including the Humboldt Foundation Senior Distinguished Scientist Award for excellence in research and teaching, the AIAA Information Systems Award and Medal for "fundamental and pioneering contributions towards the design, evaluation, and validation of dependable aerospace computing systems," and the IEEE Emanuel R. Piore Award "for fundamental contributions to measurement, evaluation, and design of reliable computing systems.