Critical Variable Recomputation for Transient Error Detection

Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer, *Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign.*

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.*

***Index Terms*— Static Analysis, Application-aware, Path-tracking, Backward Slicing, Reliability. Diverse Execution.**

I. 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. These 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, such as pointer errors and timing and synchronization errors.

Many 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 subtle errors (such as timing and synchronization errors) [8], 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: preempt uncontrolled system crash/hang and prevent error propagation.

Duplication has traditionally been used to provide high-coverage at runtime for software errors and hardware-errors. However, in order to prevent error-propagation and preempt crashes, duplication needs to make a comparison after every instruction, which in turn results in high performance overhead. Therefore, duplication approaches compare the results of replicated instructions at selected program points such as stores and branches [4]. While this reduces the performance overhead, it sacrifices coverage as the program may crash before reaching the comparison point. Further, duplication-based techniques offer limited protection from software errors and permanent hardware faults because the original program and the duplicated program can suffer from common mode errors.

Diverse execution techniques that execute two different versions of the same program, can offer protection from common-mode errors. ED4I [5] 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 certain kinds of hardware errors in processor functional units and memory. However, ED4I cannot detect errors in instruction issue and decode logic as it introduces diversity only in the data values used in the program and not in the instructions that compute the data values.

The approach presented in this paper derives runtime error detectors (or checks) based on static analysis. 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 derives runtime error detectors based on application properties extracted using static analysis. The derived detectors recompute the values of critical program variables in a diverse way in order to minimize the possibility of common-mode errors. The derived detectors are implemented using a combination of programmable hardware and software.

II. Approach

This section presents an overview of the detector derivation approach. The approach is based on program slicing [6].

Manuscript received March 23, 2007. This work was supported in part by National Science Foundation (NSF) grants CNS-0406351 and CNS-0524695, the Gigascale Systems Research Center (GSRC/MARCO), Motorola Corporation as part of the Motorola Center for Communications (UIUC), and Intel Corporation.

A. Terms and Definitions

Backward Program Slice of a variable at a program location is defined as the set of all program statements/instructions that can affect the value of the variable at that program location [6]. Slicing techniques can be static or dynamic in nature.

*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.

B. Slicing Algorithm

The slicing algorithm used 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. This is because at least one of the detectors placed in the program will be able to detect the error.

C. 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 [3]. This analysis is carried out on a per-function basis in the program i.e. each function in the program is considered separately for identification of critical variables.

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 in order to choose the checking expression for that specific control path.

Runtime checking in hardware and software:

The control path followed is tracked by the inserted instrumentation in hardware at runtime. The path-specific inserted checks are executed at appropriate points in the execution depending on the runtime control path.

The checks recompute the value of the critical variable for the runtime control path. The recomputed value is compared with the original value computed by the main program. In case of a mismatch, the original program is stopped and recovery is initiated. Otherwise, execution continues normally.

D. Hardware 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 can be significant (4x) when done in software. Therefore, a prototype implementation of path tracking is presented in hardware. This hardware is integrated with the Reliability and Security Engine (RSE) [10]. 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 framework 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.

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 IV.

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 implementation, the checking is done in software.

E. Fault Model

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

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

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

III. COMPILER-BASED DETECTOR DERIVATION

The LLVM compiler [7] is used for the analysis and derivation of error detectors. The derivation of detectors is done by introducing a new pass into LLVM, called the *Value Recomputation Pass (VRP).* The VRP performs the backward slicing starting from the instruction that computes the value of the critical variable to the beginning of the function. It also performs check derivation, insertion and instrumentation. The output of the pass is provided as input to other optimization passes in LLVM. *By extracting the path-specific backward slice and exposing it to other optimization passes in the compiler, the Value Recomputation pass enables aggressive compiler optimizations to be performed on the slice that would not be possible otherwise.*

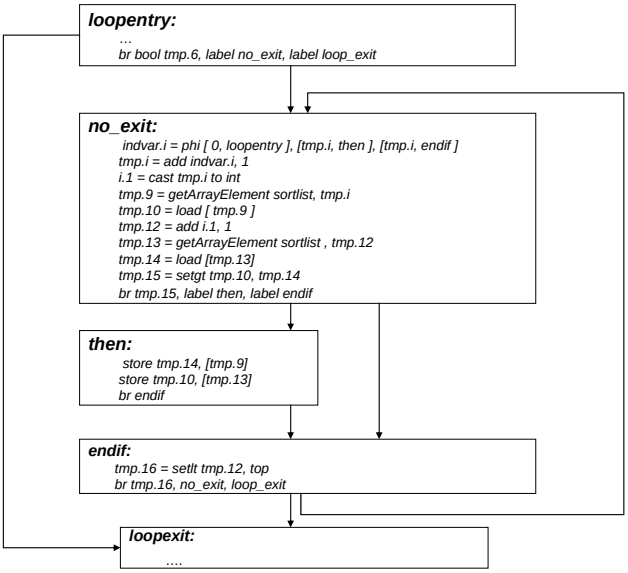

Figure 1: LLVM intermediate code corresponding to inner while loop of a Bubble sort program

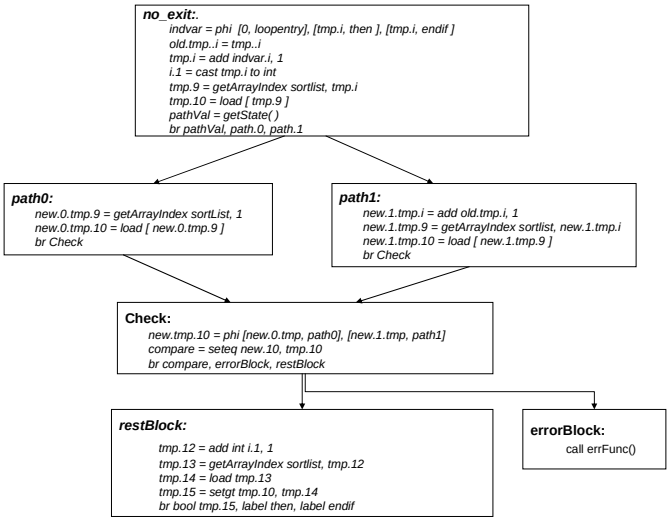

Figure 2: Transformations introduced by the VRP and other optimization passes for code shown in Figure 1.

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

A. Algorithm

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

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 and one for the critical variable.
- *Memory Dependences not considered.* While LLVM does not represent memory objects in SSA form, it promotes most memory objects to registers prior to running a pass (including the *Value Recomputation pass).* Since there is an unbounded number of virtual registers for storing variables in SSA form, the compiler is not constrained by the number of physical registers.

The details of the VRP are omitted due to space constraints. The VRP algorithm may be found in the technical report version of this paper [11].

B. Derived Checks

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

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

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

C. Discussion

As illustrated in the example in Figure 2, 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 1 (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 2). In *path0,* the compiler performs constant propagation and replaces the computation of *tmp.i* with the constant *1* in Figure 2. 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 2). 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 2). 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 (Figure 2). 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 VRP 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.

IV. HARDWARE-BASED PATH TRACKING

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 II.D, the path-tracking hardware is implemented as a module in the RSE [10] 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.

A. 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.

B. Module Components

The path-tracking module is shown in Figure 3. 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.

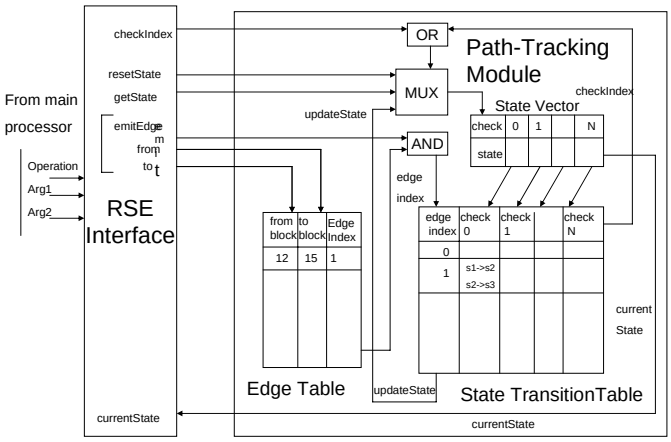

Figure 3: Hardware module for tracking paths

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

C. Module Operation

The operation of the path-tracking module for each of the following operations (executed in the processor) is considered:

CHECK instruction with emitEdge operation:

- 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:

- 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 resetState operation: This is similar to the *getState* operation, except that no value is returned.

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.

V. 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.

Future work will involve evaluating the performance overhead and error detection coverage of the derived detectors. We also plan to implement the checking expressions derived in hardware and joint synthesis of the path-tracking module.

REFERENCES

[1] D. Evans, J. Guttag, 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), Dec. 1994.

[2] 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.

[3] 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

[4] 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.

[5] N.S. Oh, S. Mitra and E.J. McCluskey, *ED4I: Error Detection by diverse data and duplicated instructions in super-scalar processors*. IEEE Transactions on Computers, 51(2):pp. 180-199, February 2002.

[6] Mark Weiser. *Program slicing*. In Proceedings of the 5th International Conference on Software Engineering, pages 439--449. IEEE Computer Society Press, 1981.

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

[8] 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 (SRDS), pages 3--12, 1986

[9] 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 (TOPLAS),13(4) 1991 pp.451–490.

[10] N. Nakka, J.Xu, Z.Kalbarczyk, and 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.

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