

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(Urbana-Champaign)

{pattabir, kalbar, [rkiyer](mailto:rkiyer@uiuc.edu)}@uiuc.edu

1. Introduction

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

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 software 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, duplication suffers from the following disadvantage: in order to prevent error-propagation and preempt crashes, a comparison needs to be performed after every instruction, which in turn results in high performance overhead. Therefore, duplication approaches compare the results of replicated instructions at selected program points such as stores to memory [4]. 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 offer limited protection from software faults and permanent hardware faults because the original program and the duplicated program can suffer from common mode errors. In order to deal with common-mode errors, *diverse execution techniques* that execute two different versions of

the same program and compare the results must be used.

ED4I [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 hardware errors in processor functional units and memory. However, ED4I cannot detect software errors that result in incorrect computation of data values in both the original program and the transformed program.

The approach presented in this paper derives runtime error detectors (or checks) based on static analysis for critical variables in the program. Critical variables are variables that are highly sensitive to data errors, and deriving detectors for such variables ensures high coverage. The derived detectors recompute the value of critical variables without replicating the entire program. Our technique introduces diversity in the computation of critical variables and is consequently able to detect both hardware errors and certain kinds of software errors.

1.1 Fault Model

The technique presented in this paper considers transient errors in data due to both hardware faults and software faults as follows:

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

- *Processor data path:* Includes errors in functional units or in the register file that result in data-value corruption.
- *Processor control path:* Includes errors in the instruction decode/issue units that result in the wrong instruction being executed.

Memory/Cache: Errors in the memory or cache caused due to cosmic radiation and disturbances.

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

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

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

2. Approach

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

2.1 Terms and Definitions

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

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

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

2.2 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.

2.3 Steps in Detector Derivation

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

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

2.4 Implementation

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.*

3. Example

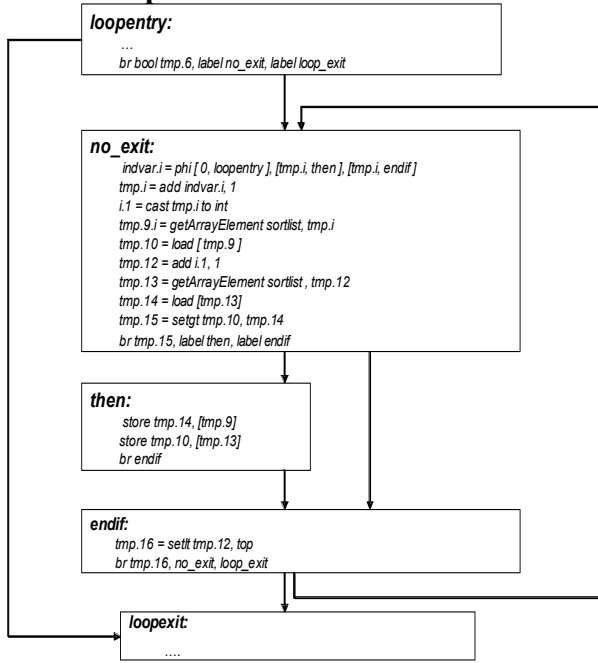


Figure 1: LLVM intermediate code corresponding to inner while loop of a Bubble sort program

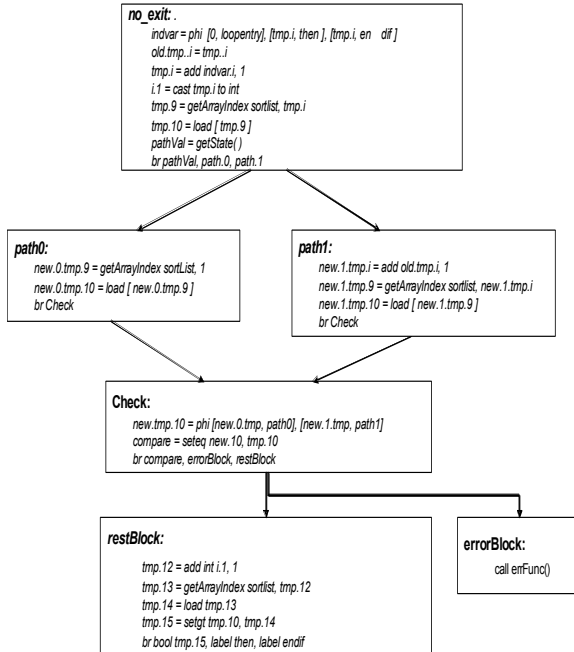


Figure 2: Transformations introduced by the VRP and other optimization passes for LLVM code shown in Figure 1.

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 data dependences among instructions.

Assume that the variable *tmp.10* has been identified as a critical variable. The final outcome after running the VRP and optimization passes is shown in Figure 2 for the computation of the critical variable *tmp.10*. The backward slice of this variable consists of the instructions that compute the values of *tmp.9*, *tmp.i*, *indvar.i*. These instructions are specialized depending on the control path followed in the program. The details of the VRP are not presented in this paper due to space constraints, but may be found in [10]. *This paper focuses on the errors detected by the derived detectors.*

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.

3.1 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 $new.1.tmp.i$ 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 $new.1.tmp.10$ 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.*

4. Conclusions

This paper presented a technique to derive error detectors for protecting an application from data errors. The error detectors are derived automatically using compiler-based static analysis from the backward program slice of critical variables in the program. The slice is optimized aggressively 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 expressions corresponding to the control-path followed are executed. The checking expression recomputes the value of the critical variable and a mismatch between the recomputed and original values indicates an error. The diversity introduced by the checking expression allows it to detect both software and hardware errors in the application.

5. 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), December 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.S. 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 Reliability, 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 ACM Symp. on Code Generation and Optimization (CGO'04), Palo Alto, CA, 2004.
- [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, 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 13(4) 1991 pp.451-490.
- [10] K. Pattabiraman and Ravishankar Iyer, *Automated Derivation of Application-aware Error Detectors using Compiler Analysis*, under submission to DSN 2007.