SymPLFIED: Symbolic Program Level Fault-Injection and Error Detection Framework

Karthik Pattabiraman
Nithin Nakka
Zbigniew Kalbarczyk
Ravishankar Iyer

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign (UIUC)
Latent errors ➔ Propagation ➔ Failures

- Errors may not manifest for billions of cycles [Guo3]
  - Program may perform harmful actions in this time
  - Error may propagate and corrupt permanent state

Error detection mechanisms vital for reliability

- Need to validate detection mechanisms
- Improve coverage based on insights learned
Formal Verification is a complementary technique to traditional fault-injection that can expose corner-case scenarios.
Need for a formal framework to evaluate the effects of hardware errors on arbitrary programs independent of the detection mechanism.
Main Contributions

- **Formal model for analyzing programs**
  - Evaluate effect of transient hardware errors directly on programs written in assembly language

- **Specification of error detectors in same model**
  - Compare detectors and understand their limitations

- **Symbolic error propagation technique**
  - Represent errors using a single symbol (err) to avoid state space explosion
Outline

- Introduction
- Approach
- Examples
- Implementation
- Case Study
SymPLFIED: Approach

- Analyze program in **assembly language**
  - Low-level state made explicit to analysis
  - Post compile-time optimizations
  - Both programs and libraries

- **Generic representation of error detectors**
  - CHECK instructions inline with program

- **Fault Model: Transient Hardware Errors**
  - **Computation**: Instruction Fetch/Decode/Execute
  - **Memory elements**: Registers/Memory, Bus Errors
Model error propagation by representing all error values in program as abstract symbol ($\text{err}$)
- Avoids state space explosion of exhaustive injection
- States identified based on where errors propagate to
- Represents both single- and multi-bit errors

Assumption: Completeness more important than accuracy as detectors can be conservative
- Constraint solver to minimize false-positives
Outline

- Introduction
- Approach
- Examples
- Implementation
- Case Study
**Example: Error Propagation**

Symbolic propagation can lead to comprehensive enumeration of program outcomes due to the injected error.

```
ori $2 $0 #1
read $1
mov $3, $1
ori $4 $0 #1

loop: setgt $5 $3 $4

beq $0 #0 loop

exit: prints "Factorial = "
print $2
halt

($3 > $4) = true
Exit loop

($3 > $4) = false
Reenter loop and continue

true => \{N!, (N!/2!), (N!/3!) \ldots 1\}
false => err
```
Example: Error Detection

Symbolic propagation can elucidate errors that escape detection and the circumstances under which they escape.

ori $2 $0 #1
read $1
mov $3, $1
ori $4 $0 #1

loop: setgt $5 $3 $4
beq $5 0 exit

check ($3 > $4)
mov $6, $2
mult $2 $2 $3
check ($2 <= $6 * $1)
subi $3 $3 #1
beq $0 #0 loop

exit: prints "Factorial = 
print $2
halt

($3 > $4) == false
Exit loop

($2 <= $6 * $1) == true,
if (1 <= $3 <= $1 )

true

($2 <= $6 * $1) == false,
if (1 <= $1 <= $3 )
Outline

- Introduction
- Approach
- Examples
- Implementation
- Case Study
SymPLFIED: Implementation

- **Rewriting Logic**
  - **Equations** for modeling **deterministic** actions
    - E.g. Instruction execution, memory/register file lookups
  - **Rewrite rules** for **non-deterministic** transitions
    - E.g. Branch on error condition, address lookup with errors

- **Implemented using Maude tool (UIUC/SRI)**
  - Executable specification in rewriting logic
  - Automated translation from assembly code
  - Exhaustive search (**bounded model checking**)
SymPLFIED : Framework

SymPLFIED Modules
(Rewriting logic)

Machine Model
(Memory, Registers, Instructions)

Error Model
(Register errors, memory errors, control-flow errors)

Detector Model
(Specification and execution of error detectors)

Modular framework allows decoupling of detection mechanism and error class from the execution model and verification technique

User supplied

Comprehensive enumeration of all errors missed by detectors that cause the program to fail
Outline

- Introduction
- Approach
- Examples
- Implementation
- Case Study
Case Study - Tcas

- **Application Characteristics**
  - Aircraft collision avoidance system
  - Used by majority of aircraft in the USA (FAA)
  - Rigorously specified and verified protocol

- **Code characteristics (Siemens suite)**
  - About 150 lines of C code = 900 lines of assembly
  - **Inputs**: aircraft position parameters
  - **Outputs**: 0 = Un-resolvable, 1 = Ascend, 2 = Descend
Considered a case where the output of tcas was 1 (ascend) and injected all possible register errors

- Found 1 potentially catastrophic outcome
  - Output of 2 (descend) instead of 1 (ascend)

- Highly-parallelizable code of SymPLFIED
  - Took about 4 minutes on a 300 node cluster
  - Total of 1200 minutes of machine time
## Fault-Injection Results (SimpleScalar Simulator)

<table>
<thead>
<tr>
<th>Program Outcome</th>
<th>Percentage of errors</th>
<th># faults = 6253</th>
<th># faults = 41082</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Undecidable)</td>
<td>1.86% (117)</td>
<td>2.33% (960)</td>
<td></td>
</tr>
<tr>
<td>1 (Ascend)</td>
<td>53.7% (3364)</td>
<td>56.33% (23143)</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>0.5% (29)</td>
<td>1.0% (404)</td>
<td></td>
</tr>
<tr>
<td>Crash</td>
<td>43.4% (2718)</td>
<td>40.43% (16208)</td>
<td></td>
</tr>
<tr>
<td>Hang</td>
<td>0.4% (25)</td>
<td>0.8% (327)</td>
<td></td>
</tr>
</tbody>
</table>

**Not a single case** of the catastrophic outcome observed, when
(1) Simulations ran for same time as symPLFIED
(2) Simulations ran for four times as much time
int alt_sep_test()
{
    ...

    alt_sep = UNRESOLVED;

    if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped)) {

    Non_Crossing_Biased_Climb()
    &&
    Own_Below_Threat();

    need_downward_RA =
    Non_Crossing_Biased_Descend()
    &&
    Own_Above_Threat();

    if (need_upward_RA && need_downward_RA)
        alt_sep = UNRESOLVED;
    else if (need_upward_RA)
        alt_sep = UPWARD_RA;
    else if (need_downward_RA)
        alt_sep = DOWNWARD_RA;

    }

    return alt_sep;

(1) Assembly-language level reasoning needed to expose runtime error
(2) Random injection needs to get both type and location of fault right

alt_sep = UNRESOLVED;
else if (need_upward_RA)
    alt_sep = UPWARD_RA;
else if (need_downward_RA)
    alt_sep = DOWNWARD_RA;

Non_Crossing_Biased_Climb:
Return address in register $31
corrupted by symbolic error
Conclusions and Future Work

- SymPLFIED: Formal program-level fault-injection and error-detection framework
  - Directly works on assembly language programs
  - Uses model-checking and symbolic execution
  - Comprehensively enumerate missed errors

- Ongoing/Future Directions
  - Probabilistic evaluation of detection coverage
  - Compositional Reasoning (for scalability)
  - Extension to error classes and architectures