An End-to-End Approach for the Automatic Derivation of Application-Aware Error Detectors

Galen Lyle
Shelley Chen
Karthik Pattabiraman
Zbigniew Kalbarczyk
Ravishankar Iyer
Motivation: Error Detection

• Existing techniques for error detection (e.g. duplication)
  • Incur substantial overheads in space or time (60 to 100 %)
  • Detect errors that do not impact application (benign errors)

• Goal: Detect errors that matter to the application before propagation (at low cost)

• Our Approach: Application-aware Detection
  • Error detectors based on properties of the application
  • Automated process for deriving and implementing error detectors
Earlier Work: Critical Variable Recomputation (CVR)

- **Insertion of runtime checks in the program**
  - Automatic analysis through new compiler pass
  - Checks synthesized as custom software libraries

- **Coverage evaluation using fault-injections**
  - Detection coverage = 77 % (for failure-causing errors)
  - Less than 3 % benign error detections

- **But, performance overheads ….**
  - Up to 141 % on Pentium-4 processor (average = 33 %)
  - Up to 555 % on Leon-3 (due to limited parallelism)
This Paper: H/W Implementation

- Hardware implementation to reduce overheads
  - Monitoring/checking in parallel with application

- Implemented using Reconfigurable Hardware (FPGAs)
  - Application executes on general-purpose processor
  - Checks execute on separate module - **Static Detector Module**
  - Module interfaces with processor through generic interface (RSE)

---

FPGA

![Diagram showing General-purpose Processor, RSE, Static Detector Module (SDM)]

University of Illinois at Urbana-Champaign (UIUC)
Paper Outline

• Motivation and Approach

• CVR Technique (Recap)

• H/W Implementation

• Experimental Results

• Conclusions and Future Work
CVR Technique: Overall Algorithm

1. Identify critical variables thro’ profiling
   [PRDC’05]

2. Extract backward slice of critical variable thro’ static analysis
   [IOLTS’07][TDSC in press]

3. Specialize slice according to control-paths and optimize to obtain checking expression

4. Instrument program to track runtime paths and execute the corresponding checks

University of Illinois at Urbana-Champaign (UIUC)
CVR Technique: Example

Original Code

```
if (path==1)
    f2 = 2c – e;
if (a==0)
    f2 = a + e;
if (a != 0)
    if (f2==f)
        then
            Critical f;
        else
            else
```

```c
b = a + c;
d = b – e;
f = d + b;
c = a – d;
b = d + e;
f = b + c;
```

Path tracking

Rest of code

```
if (a == 0)
    Critical f;
else
    else
```

Derived Checks

```
if (path==1)
    f2 = 2c – e;
    if (a==0)
        then
            Critical f;
        else
            else
```

```c
f2 = a + e;
if (a != 0)
    then
        path=1
    else
        path=2
```

```
then
    then
if (f2==f)
    then
        Declare Error in f along path
    else
        else
```

University of Illinois at Urbana-Champaign (UIUC)
Paper Outline

- Motivation and Approach
- CVR Technique (Recap)
- H/W Implementation
- Experimental Results
- Conclusions and Future Work
H/W Implementation: Design Flow

Application source code + profile

Enhanced Compiler (CVR algorithm)

Software

Application code instrumented with special CHK instructions

Path-tracking state machines

Checking Expressions

Hardware

Regular compiler

Translation to VHDL code

General-purpose Processor

Static Detector Module (SDM)
H/W Implementation: Path Tracking

Control-flow Graph

1

2

emitEdge(1,2)

3

4

5

emitEdge(5,6)

6

7
critical variable check

Path-tracking State Machine

A

(4,6)

B

(2,7)

G

(5,6)

(1,2)

S

C

D

E

H

(7,2)

(4,6)

(2,7)

(5,6)

F

University of Illinois at Urbana-Champaign (UIUC)
H/W Implementation: Checking Expression

Inputs: Passed in sharedARGQ
   sortlist
   indvar.i.copy

Instructions: Converted to DAG
   t0 = 1
   t1 = sortList
   t2 = getArrayElement t0, t1
   t3 = load (*t2)
   t4 = indvar.i.copy
   t5 = t0 + t4
   t6 = getArrayElement t5, t1
   t7 = load (*t6)

Path 0: Final = t3
Path 1: Final = t7
H/W Implementation: Design Flow

Application source code + profile

Enhanced Compiler (CVR algorithm)

Application code instrumented with special CHK instructions
Path-tracking state machines
Checking Expressions

Software

Regular compiler

Hardware

Translation to VHDL code

General-purpose Processor

Static Detector Module (SDM)
H/W Implementation: SDM

- **Static Detector Module (SDM)**
  - Path-tracking
  - Check execution

- **Path-tracking sub-module**
  - Monitors $emitEdge$ operations
  - State Stack for function calls

- **Checking sub-module**
  - Parameter passing: ARGQ
  - Direct access to memory

University of Illinois at Urbana-Champaign (UIUC)
H/W Implementation: RSE

Leon 3 Pipeline

- Fetch
- Decode
- Register
- Execute
- Memory
- Exception
- Writeback

Leon 3 System Bus

RSE Interface

- Controller
- Static Detector Module
- Other Module
- Other Module
- DMA Controller

RSE Modules

Leon 3 System Bus
Paper Outline

- Motivation and Approach
- CVR Technique (Recap)
- H/W Implementation
- Experimental Results
- Conclusions and Future Work
Experimental Setup

- **Benchmarks: Stanford Programs**
  - 100-500 lines of C code

- **VHDL generated according to check and path definitions obtained during compilation**
  - VHDL simulation in ModelSim 6.3
  - Hardware synthesis for Xilinx Virtex-2 Pro FPGA

- **Measurement of area and performance overheads using direct execution on FPGA hardware**
  - Includes overheads of RSE interface (but no other modules)
Results (Area Overheads)

Area Overhead of Checking Hardware on Leon3

Area Increase (%)

Benchmark

Overheads likely to be lower in the case of more complex processors

Bigger checks

Avg = 53 %

University of Illinois at Urbana-Champaign (UIUC)
Results (Performance Overheads)

Additional Cycles Required in Execution

Overhead of emitEdge instructions at branch points

Significant reduction compared to software (200-600) %

Avg = 27 %

Bubblesort, IntMM, Oscar, Perm, Queens, Quicksort, RealMM, Mean

University of Illinois at Urbana-Champaign (UIUC)
Paper Outline

- Motivation and Approach
- CVR Technique (Recap)
- H/W Implementation
- Experimental Results
- Conclusions and Future Work
Conclusions

- Application-aware detection is a low-overhead approach for providing high detection coverage
  - Compiler support to derive detectors (checks)
  - Reconfigurable hardware support for executing checks

- H/W Implementation as RSE module
  - Synthesized and implemented on FPGA
  - Area overheads bet. 50-55 % (Avg = 53 %)
  - Perf. Overheads bet. 15-65 % (Avg = 27 %)

Low performance overheads with SDM, but potentially unbounded area overheads
Ongoing work: Micro-controller

- **Our idea**
  - Use a small micro-controller in place of the SDM

- **Programmed at application load time**
  - No need for reconfiguration

- **Results**
  - Perf overhead = 30 % (Vs. 27 %)
  - Area overhead = 45 % (const)
Future Work

- Feedback loop from hardware to compiler for estimating performance and area overhead apriori.

- Eliminate \textit{emitEdge} instructions by observing branches (reduction of performance overheads).

- Identify common state across checks and optimize these out (reduction of area overhead).
Trusted ILLIAC Project (UIUC)

Custom hardware as extension cards

Trusted Illiac Node for prototyping and development

Projected Vision
• 256 nodes

University of Illinois at Urbana-Champaign (UIUC)
Thank You

Questions?

Contact: Karthik.Pattabiraman@gmail.com