Automated Bug Detection for Pointers and Memory Accesses in High-Level Synthesis Compilers

Pietro Fezzardi  
pietro.fezzardi@polimi.it

Fabrizio Ferrandi  
fabrizio.ferrandi@polimi.it

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milano, Italy

FPL 2016 – Lausanne – 01/09/2016
Outline

Introduction and Motivation

Background and Assumptions

Automated Bug Detection for Pointers

Evaluated Tools, Experiments and Results

Conclusion and Future Work
Motivation

Adoption of High-Level Synthesis is increasing
HLS tools are becoming increasingly complex
Memory optimizations bring substantial improvements
Memory bugs introduced by HLS tools are hard to debug

A methodology to automatically find memory bugs introduced by the compiler would:

- Make existing memory allocation and optimizations more reliable
- Ease development and deployment of new memory architectures in HLS
- Speed up testing of new memory optimizations in HLS
- Make easier for HLS developers and users to isolate the cause of bugs
Goals

General Ideas

- Take advantage of HLS information to support all compiler optimizations
- Automatically isolate the wrong signal, failing operation and component
- Automatically backtrack the error to the original source code
- Avoid user interaction to enable massive automated testing in production

Goals Related to Pointers

- Specifically target memory bugs involving pointers and addresses
- Completely support C standard pointer based descriptions
- Support different memory technologies and partitioning patterns
- Independent of memory optimizations
Discrepancy Analysis Debug Flow

ORIGINAL SOURCE CODE (C) → FRONTEND → CONTROL DATA FLOW GRAPH (CDFG) → HLS → FINITE STATE MACHINE (FSM) → BACKEND → HARDWARE DESCRIPTION LANGUAGE (HDL)

**EXTRACTED INFORMATION**
- Source level information
- Instrumented C code in SSA after frontend optimizations
- Memory layout
- Memory allocation
- Memory partitioning

**FSM optimizations**
- Chaining
- Pipelining
- Resource sharing
- Scheduling
- Binding

**List of signals to collect**
- HW traces

**DEBUGGING STEPS**
- Compilation + execution
- **SW Traces**
  - Address operations
  - SW memory locations
  - SW Call Context IDs
- **ASTS**
- **RTL simulation**
- HW Traces (VCD)

**DISCREPANCY ANALYSIS**
- Debug information on first mismatch between HW and SW executions
- Faulty module in HW design
- State of FSM
- Hierarchical path and name of wrong signal
- Value of wrong signal
- Failed operation in C
- Start and end time of the failed operation
- Value of related C variable
- SW call stack trace

**P.Fezzardi – pietro.fezzardi@polimi.it**
Discrepancy Analysis Debug Flow

ORIGINAL SOURCE CODE (C) → FRONTEND → CONTROL DATA FLOW GRAPH (CDFG) → HLS → FINITE STATE MACHINE (FSM) → DATA PATH (DP) → BACKEND → HARDWARE DESCRIPTION LANGUAGE (HDL)

EXTRACTED INFORMATION:
- Source level information
- Instrumented C code in SSA after frontend optimizations
- FSM optimizations
- Chaining
- Pipelining
- Resource sharing
- Scheduling
- Binding
- Allocation
- List of signals to collect HW traces

DEBUGGING STEPS:
- Compilation + execution

SW Traces of:
- Address operations
- SW memory locations
- SW Call Context IDs

ASTS

HW Traces (VCD)

DISCREPANCY ANALYSIS DEBUG INFORMATION:
- First mismatch between HW and SW executions
- Faulty module in HW design
- State of FSM
- Hierarchical path and name of the wrong signal
- Value of the wrong signal
- Failed operation in C
- Start and end time of the failed operation
- Value of the related C variable
- SW call stack trace

P. Fezzardi – pietro.fezzardi@polimi.it

FPL 2016 – Lausanne – 01/09/2016
Discrepancy Analysis Debug Flow

- **ORIGINAL SOURCE CODE (C)**
- **FRONTEND**
- **CONTROL DATA FLOW GRAPH (CDFG)**
- **HLS**
- **FINITE STATE MACHINE (FSM)**
- **DATA PATH (DP)**
- **BACKEND**
- **HARDWARE DESCRIPTION LANGUAGE (HDL)**

**EXTRACTED INFORMATION**
- Source level information
- Instrumented C code in SSA after frontend optimizations
- Compilation and execution
- SW Traces

**FSM optimizations**
- Chaining
- Pipelining
- Resource sharing
- Scheduling
- Binding
- Allocation
- List of signals to collect HW traces

**DEBUGGING STEPS**
- Compilation + execution
- SW Traces

**SW Traces**
- SW Traces of address operations
- SW memory locations
- SW Call Context IDs

**HW Traces (VCD)**
- Cycle accurate RTL simulation

**DISCREPANCY ANALYSIS**
- First mismatch between HW and SW executions
- Faulty module in HW design
- State of FSM
- Hierarchical path and name of the wrong signal
- Value of the wrong signal
- Failed operation in C
- Start and end time of the failed operation
- Value of the related C variable
- SW call stack trace

P. Fezzardi – pietro.fezzardi@polimi.it
Discrepancy Analysis Debug Flow

ORIGINAL SOURCE CODE (C)
FRONTEND
CONTROL DATA FLOW GRAPH (CDFG)
HLS
FINITE STATE MACHINE (FSM)
DATA PATH (DP)
BACKEND
HARDWARE DESCRIPTION LANGUAGE (HDL)

EXTRACTED INFORMATION

source level information
instrumented C code in SSA after frontend optimizations

DEBUGGING STEPS

SW Traces
HW Traces (VCD)

SW Traces of
address operations
SW memory locations
SW Call Context IDs

ASTS

cycle accurate RTL simulation

HW Traces (VCD)

list of signals to collect HW traces

FSM optimizations
chaining
pipelining
resource sharing
scheduling
binding
allocation

P.Fezzardi – pietro.fezzardi@polimi.it
FPL 2016 – Lausanne – 01/09/2016
Discrepancy Analysis Debug Flow

ORIGINAL SOURCE CODE (C) → FRONTEND → CONTROL DATA FLOW GRAPH (CDFG) → HLS → FINITE STATE MACHINE (FSM) → DATA PATH (DP) → BACKEND → HARDWARE DESCRIPTION LANGUAGE (HDL)

EXTRACTED INFORMATION:
- source level information
- instrumented C code in SSA after frontend optimizations
- compilation + execution
- FSM optimizations
- chaining
- pipelining
- resource sharing
- scheduling
- binding
- allocation
- list of signals to collect HW traces
- cycle accurate RTL simulation

SW Traces → DISCREPANCY ANALYSIS → HW Traces (VCD)

DEBUGGING STEPS:
- compilation + execution
- SW Traces
- cycle accurate RTL simulation

DISCREPANCY ANALYSIS:
- first mismatch between HW and SW executions
- faulty module in HW design
- state of FSM
- hierarchical path and name of the wrong signal
- value of the wrong signal
- failed operation in C
- start and end time of the failed operation
- value of the related C variable
- SW call stack trace
Discrepancy Analysis Debug Flow

**ORIGINAL SOURCE CODE (C)**
- compile
- execution

**FRONTEND**
- control data flow graph (CDFG)
- HLS

**FINITE STATE MACHINE (FSM)**
- FSM optimizations
- chaining
- pipelining
- resource sharing
- scheduling
- binding
- allocation

**DATA PATH (DP)**
- memory layout
- memory allocation
- memory partitioning

**BACKEND**
- hardware description language (HDL)

**INSTRUMENTED C CODE IN SSA AFTER FRONTEND OPTIMIZATIONS**

**EXTRACTED INFORMATION**

**DEBUGGING STEPS**

**DEBUG INFORMATION**
- first mismatch between HW and SW executions
- faulty module in HW design
- state of FSM
- hierarchical path and name of the wrong signal
- start and end time of the failed operation
- value of the wrong signal
- value of the related C variable
- failed operation in C
- SW call stack trace

**SW Traces**

**HW Traces (VCD)**

**DISCREPANCY ANALYSIS**
- list of signals to collect HW traces
- cycle accurate RTL simulation
- SW Traces of address operations
- SW memory locations
- SW Call Context IDs
- ASTS

P. Fezzardi – pietro.fezzardi@polimi.it
Memory Locations

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>In HW</th>
<th>In SW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle M_i, B_i, S_i \rangle$</td>
<td>○ $M_i$: unique identifier for a memory module</td>
<td>○ $M_i$: can be omitted</td>
</tr>
<tr>
<td></td>
<td>○ $B_i$: an offset in the memory module identified by $M_i$</td>
<td>○ $B_i$: address in main memory</td>
</tr>
<tr>
<td></td>
<td>○ $S_i$: size of the memory location</td>
<td>○ $S_i$: size of the memory location</td>
</tr>
</tbody>
</table>

Similar to **Location Sets** [Wilson and Lam; PLDI’95] [Séméria and De Micheli; TCAD’01]

Abstract concepts, independent of the target memory architecture

HW Memory Locations are not addresses but can be directly translated to addresses

Evaluated HLS compilers (Bambu, LegUp, Commercial Tool) use equivalent representations
For memory allocation, HLS tools take mainly two decisions:

**which variables have to be allocated in memory**

- usually global, static, volatile, arrays, and structs
- possibly others, according to alias analysis

**the location where every memory-mapped variable is stored**

- depends on HLS implementation
- depends on memory architecture of the generated design
- depends on the memory optimizations and partitioning
Assumptions for Address Discrepancy Analysis

**General Assumption I**
Every HW Memory Location must be associated to a single memory-mapped variable

The inverse mapping of high-level variables onto HW Memory Locations must be known

It is simply the inverse of the mapping computed by memory allocation in HLS ✓

**General Assumption II**
It has to be possible to identify the signals representing pointer variables in HW

Previous results show that this is possible ✓
### Address Space Translation Scheme (ASTS)

<table>
<thead>
<tr>
<th>Hardware Address Table (HAT)</th>
<th>Software Address Table (SAT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i \langle M_i, B_i, S_i \rangle$</td>
<td>$j \langle CB_i, CS_i \rangle$</td>
</tr>
</tbody>
</table>

$i$: variable identifier  
$\langle M_i, B_i, S_i \rangle$: HW Memory Location  

$j$: SW Call Context ID  
$\langle CB_i, CS_i \rangle$: SW Memory Location  

$i$: variable identifier
The Software Call Context Identifier

In HW
The HAT is computed by memory allocation during HLS
Memory Locations in HW are defined once ahead of time

In SW
Local variables are allocated on the stack
Different Memory Location at every function call
An ID is necessary to distinguish between calls
An ID uniquely identifies a path on the call graph
Function calls are instrumented in the C code
Context ID and memory mapping are printed at runtime
The Extended Address Discrepancy Analysis Flow

**ORIGINAL SOURCE CODE (C)**

**FRONTEND**
- source level information

**CONTROL DATA FLOW GRAPH (CDFG)**
- instrumented C code in SSA after frontend optimizations

**HLS**
- FSM optimizations
  - chaining
  - pipelining
  - resource sharing

**FINITE STATE MACHINE (FSM)**
- scheduling
- binding
- allocation

**DATA PATH (DP)**
- list of signals to collect HW traces

**BACKEND**
- cycle accurate RTL simulation

**HARDWARE DESCRIPTION LANGUAGE (HDL)**

**EXTRACTED INFORMATION**

**DEBUGGING STEPS**
- compilation + execution

**SW Traces**
- SW Traces of pointer operations
- SW memory locations
- SW Call Context IDs

**HW Traces (VCD)**
- HW Traces

**DISCREPANCY ANALYSIS**
- first mismatch between HW and SW executions
- faulty module in HW design
- state of FSM
- hierarchical path and name of the wrong signal
- value of the wrong signal
- failed operation in C
- value of the related C variable
- SW call stack trace

**DEBUG INFORMATION**
- start and end time of the failed operation

**EXTRACTED INFORMATION**

**SOURCE LEVEL INFORMATION**
- memory layout
- memory allocation
- memory partitioning

**FSM OPTIMIZATIONS**
- chaining
- pipelining
- resource sharing

**SCHEDULING**
- binding
- allocation

**RESOURCE SHARING**
- list of signals to collect HW traces

**Allocations**
- list of signals to collect HW traces

**SW Traces**
- cycle accurate RTL simulation
The Extended Address Discrepancy Analysis Flow

ORIGINAL SOURCE CODE (C) → FRONTEND

CONTROL DATA FLOW GRAPH (CDFG) → HLS

FINITE STATE MACHINE (FSM) → DATA PATH (DP) → BACKEND

HARDWARE DESCRIPTION LANGUAGE (HDL)

EXTRACTED INFORMATION

source level information

instrumented C code in SSA after frontend optimizations

compilation + execution

memory layout

memory allocation

memory partitioning

FSM optimizations

chaining

pipelining

resource sharing

scheduling

binding

allocation

list of signals to collect HW traces

cycle accurate RTL simulation

DEBUGGING STEPS

compilation + execution

SW Traces

DISCREPANCY ANALYSIS

first mismatch between HW and SW executions

faulty module in HW design

state of FSM

hierarchical path and name of the wrong signal

value of the wrong signal

failed operation in C

start and end time of the failed operation

value of the related C variable

SW call stack trace

HW Traces (VCD)

DEBUG INFORMATION

P.Fezzardi – pietro.fezzardi@polimi.it

9 FPL 2016 – Lausanne – 01/09/2016
The Extended Address Discrepancy Analysis Flow

ORIGINAL SOURCE CODE (C)
FRONTEND
CONTROL DATA FLOW GRAPH (CDFG)
HLS
FINITE STATE MACHINE (FSM)
DATA PATH (DP)
BACKEND
HARDWARE DESCRIPTION LANGUAGE (HDL)

source level information
instrumented C code in SSA after frontend optimizations
memory layout
memory allocation
memory partitioning
FSM optimizations
chaining
pipelining
resource sharing
scheduling
binding
allocation
list of signals to collect HW traces
socket accurate RTL simulation

EXTRACTED INFORMATION

compilation + execution

DEBUGGING STEPS

SW Traces
HW Traces (VCD)

DISCREPANCY ANALYSIS

first mismatch between HW and SW executions
faulty module in HW design
state of FSM
hierachical path and name of the wrong signal
value of the wrong signal
failed operation in C
start and end time of the failed operation
value of the related C variable
SW call stack trace
The Extended Address Discrepancy Analysis Flow

- Original Source Code (C)
- Frontend
- Control Data Flow Graph (CDFG)
- HLS
- Finite State Machine (FSM)
- Data Path (DP)
- Backend
- Hardware Description Language (HDL)

Extracted Information:
- Source level information
- Instrumented C code in SSA after frontend optimizations
- Memory layout
- Memory allocation
- Memory partitioning
- FSM optimizations
  - Chaining
  - Pipelining
  - Resource sharing
- Scheduling
- Binding
- Allocation
- List of signals to collect HW traces
- Cycle accurate RTL simulation
- SW Traces of pointer operations
- SW memory locations
- SW Call Context IDs
- SW call stack trace
- HW Traces (VCD)

Debugging Steps:
- Compilation + execution
- SW Traces
- Debugging
- Discrepancy Analysis
- First mismatch between HW and SW executions
- Faulty module in HW design
- State of FSM
- Hierarchical path and name of the wrong signal
- Value of the wrong signal
- Failed operation in C
- Start and end time of the failed operation
- Value of the related C variable
- HW memory location addressed by wrong signal
- SW memory location of the wrong pointer

Debug Information:
- State of FSM
- Name of the wrong signal
- Value of the wrong signal
- Start and end time of the failed operation
- Value of the related C variable
- SW call stack trace

Source: Pietro.Fezzardi@polimi.it

9 FPL 2016 – Lausanne – 01/09/2016
The Extended Address Discrepancy Analysis Flow

1. **Source Level Information**
   - Instrumented C code in SSA after frontend optimizations

2. **Memory Information**
   - Memory layout
   - Memory allocation
   - Memory partitioning

3. **HLS Optimization**
   - FSM optimizations
   - Chaining
   - Pipelining
   - Resource sharing
   - Scheduling
   - Binding

4. **Backend**
   - List of signals to collect HW traces

5. **Debugging Steps**
   - Compilation + execution
   - SW Traces of pointer operations

6. **DISCREPANCY ANALYSIS**
   - First mismatch between HW and SW executions
   - Faulty module in HW design
   - State of FSM
   - Hierarchical path and name of the wrong signal
   - Value of the wrong signal
   - Start and end time of the failed operation
   - Value of the related C variable
   - SW call stack trace

7. **HW Traces (VCD)**

8. **SW Traces**
   - SW Traces

9. **ASTS**
   - Cycle accurate RTL simulation

10. **Control Data Flow Graph (CDFG)**

11. **BACKEND**
    - HARDWARE DESCRIPTION LANGUAGE (HDL)

12. **FRONTEND**
    - ORIGINAL SOURCE CODE (C)

P. Fezzardi – pietro.fezzardi@polimi.it

FPL 2016 – Lausanne – 01/09/2016
The Extended Address Discrepancy Analysis Flow

**EXTRACTED INFORMATION**
- source level information
- instrumented C code in SSA after frontend optimizations
- memory layout
- memory allocation
- memory partitioning
- FSM optimizations
- chaining
- pipelining
- resource sharing
- scheduling
- binding
- allocation
- list of signals to collect HW traces

**DEBUGGING STEPS**
- compilation + execution
- SW Traces of pointer operations
- ASTS
- HW Traces (VCD)

**DISCREPANCY ANALYSIS**
- first mismatch between HW and SW executions
- faulty module in HW design
- state of FSM
- wrong variable is a pointer
- SW memory location of the wrong pointer
- HW memory location addressed by wrong signal

**DEBUG INFORMATION**
- hierarchical path and name of the wrong signal
- start and end time of the failed operation
- value of the wrong signal
- value of the related C variable
- SW call stack trace

**SOURCE LEVEL CODE (C)**
- FRONTEND
- CONTROL DATA FLOW GRAPH (CDFG)
- HLS
- DATA PATH (DP)
- BACKEND
- HARDWARE DESCRIPTION LANGUAGE (HDL)

**EXTRACTED INFORMATION**
- source level information
- instrumented C code in SSA after frontend optimizations
- memory layout
- memory allocation
- memory partitioning
- FSM optimizations
- chaining
- pipelining
- resource sharing
- scheduling
- binding
- allocation
- list of signals to collect HW traces

**DEBUGGING STEPS**
- compilation + execution
- SW Traces of pointer operations
- ASTS
- HW Traces (VCD)

**DISCREPANCY ANALYSIS**
- first mismatch between HW and SW executions
- faulty module in HW design
- state of FSM
- wrong variable is a pointer
- SW memory location of the wrong pointer
- HW memory location addressed by wrong signal

**DEBUG INFORMATION**
- hierarchical path and name of the wrong signal
- start and end time of the failed operation
- value of the wrong signal
- value of the related C variable
- SW call stack trace

**SOURCE LEVEL CODE (C)**
- FRONTEND
- CONTROL DATA FLOW GRAPH (CDFG)
- HLS
- DATA PATH (DP)
- BACKEND
- HARDWARE DESCRIPTION LANGUAGE (HDL)
Address Discrepancy Algorithm

Shared Data: $\text{ASTS} = (\text{SAT}, \text{HAT})$

Input: $j$: SW Call Context ID
$s$: SW address assigned to a pointer $p$ in $j$
$h$: value of the signal related to $p$ in HW

Result: true if $s$ and $h$ mismatch, false otherwise

1. $i = \text{search} (j, s)$ in SAT;
2. if ($i$ is not found) then
   // $s$ is not in range for any variable
   return false;
3. else
   $\langle M_i, B_i, S_i \rangle = \text{search} (i)$ in HAT;
4. if ($\langle M_i, B_i, S_i \rangle$ is not found) then
   // not memory-mapped in HW
   return true;
5. else
   $h' = \text{decodeHW} (\langle M_i, B_i, S_i \rangle)$;
6. if $h \neq h'$ then
   return true;
7. else
   return false;

---

ASTS

<table>
<thead>
<tr>
<th>SAT</th>
<th>HAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j$</td>
<td>$\langle CB_i, CS_i \rangle$</td>
</tr>
<tr>
<td>$i$</td>
<td>$\langle M_i, B_i, S_i \rangle$</td>
</tr>
</tbody>
</table>

---

main memory

$\langle CB_i, CS_i \rangle$

$0x$FFFF1234

sizeof(int)

---

module_f

$0x$FFFF1234

sizeof(int)

---

module_g

$\langle M_i, B_i, S_i \rangle$

$B_i = 0x4$

$S_i =$ sizeof(int)
# Evaluated HLS Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bambu</strong></td>
<td>Developed at Politecnico di Milano, Based on GCC (4.5 up to 6), Free Software (GPLv3)</td>
</tr>
<tr>
<td><strong>Commercial Tool</strong></td>
<td>Production-ready, Recent version (late 2015 - early 2016), Targets Xilinx FPGAs, Closed source proprietary license</td>
</tr>
<tr>
<td><strong>LegUp</strong></td>
<td>Developed at University of Toronto, Based on LLVM, Free for non-commercial not-for-profit use</td>
</tr>
</tbody>
</table>
## Benchmarks

**CHStone [Hara et al.; ISCAS’08]**
- Well known benchmark suite for HLS
- Both control- and data-oriented examples
- 12 self-contained C programs
- Try to settle a common ground for HLS tools

**GCC C-torture**
- More than 800 tests from GCC test suite
- Designed to test obscure corner-cases
- Designed to stress test compilers
- Selected 216 cases to test pointers
## Test Matrix

<table>
<thead>
<tr>
<th></th>
<th>CHStone</th>
<th>GCC C-torture</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bambu</strong></td>
<td>✓</td>
<td>✓ full automated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(several bugs found)</td>
</tr>
<tr>
<td><strong>Commercial Tool</strong></td>
<td>use partitioning directives</td>
<td>56/216 failed HLS</td>
</tr>
<tr>
<td></td>
<td>manual bug insertion</td>
<td>manual on a short list</td>
</tr>
<tr>
<td></td>
<td>imitate known bugs found in Bambu</td>
<td></td>
</tr>
<tr>
<td></td>
<td>manual reconstruction of ASTS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>manual execution</td>
<td></td>
</tr>
<tr>
<td><strong>LegUp</strong></td>
<td>✓ partialy automated</td>
<td>✓ partially automated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(on a short list)</td>
</tr>
</tbody>
</table>
Significance of Pointer Operations

- adpcmO0
- adpcmO3
- aesO0
- aesO3
- bfO0
- bfO3
- dfaddO0
- dfaddO3
- dfdivO0
- dfdivO3
- dfmulO0
- dfmulO3
- dfsinO0
- dfsinO3
- gsmO0
- gsmO3
- jpegO0
- jpegO3
- mipsO0
- mipsO3
- mpegO0
- mpegO3
- shaO0
- shaO3
- C-torture

Static % of pointer operations on the total
Significance of Pointer Operations

![Graph showing the significance of pointer operations in various contexts. The x-axis represents different benchmarks and libraries, while the y-axis shows the percentage of operations. The graph compares static percentage of pointer operations on the total against the percentage of pointers with fully resolved alias analysis.]

- **Static % of Pointer Operations on the Total**
- **% of Pointers with Fully Resolved Alias Analysis**

P. Fezzardi – pietro.fezzardi@polimi.it

FPL 2016 – Lausanne – 01/09/2016
Detected Bugs

**Compiler Frontend**
Wrong static analysis or IR manipulations

*Example*
Static bit-width analysis to reduce the bits of addresses
A bug caused a wrong number of bits to be computed
Wrong values were used to address the memory

**Scheduling**
Wrong construction of the FSM

- missing dependencies
- wrong computation of execution times

*Example*
Missing information about data dependencies
Scheduling decided to compute an address in advance
Data necessary for the computation was not ready yet
Again generating wrong addresses

**Memory Allocation**
Memories with wrong ports, size, latency, etc.

- **Memory too small**
LOAD: read a corrupted data or hang
STORE: out-of-bound access

- **Memory too large**
LOAD/STORE: wrong offset calculation; data corruption

- **Wrong latency**
LOAD: use data before they are ready
STORE: release memory before data is stored

**Interconnection**
Connection of wrong modules
Wrong size of buses and other wirings

*Example*
Bug in bit-width analysis caused wrong size of address bus
Conclusions and Future Work

Conclusion
✓ Extend Discrepancy Analysis to support pointers
✓ Effectively fill a considerable blank in Discrepancy Analysis
✓ Independent of compiler optimizations, memory technology and partitioning patterns
✓ Avoid user interaction to enable massive automated testing in production
✓ Find several bugs in different compiler steps not found by normal Discrepancy Analysis

Future Work
○ Support for speculation
○ Support for synthesis of dynamic allocation malloc()/free()
Thank You for Your Attention

Questions?

Contact: pietro.fezzardi@polimi.it
Website: http://panda.dei.polimi.it
Backup Slides
int w(struct sockq *q, void *src, int len) {

    char *sptr = src;

    while (len--) {
        q->buf[q->head++] = *src);

        if (q->head == NET_SKBUF_SIZE)
            q->head = 0;
    }

    return len;
}
int main() {

    int *p, a[32], b[32], res = 0;

    for (p = a; p < a + 32; p++)
        res += something(p);

    for (p = b; p < a + 32; p++)
        res += something(p);

    return res;
}

Solution to False Positives

Address SANitizer (ASAN)

- SW memory error detector
- Deployed both in GCC (from 4.8) and LLVM (from 3.1)
  - compiler instrumentation pass
  - run-time library to replace malloc()/free()
- Adds redzones around every variable
- If a redzone is accessed triggers an error

Address Discrepancy Analysis do not check out-of-bound addresses
Wild pointers operations are allowed in C
If a wild pointer is dereferenced in C ASAN catch it
Even if out-of-bounds pointers are not checked ASAN ensures everything is ok
Performance overhead compared to simulation

- adpcmO0
- adpcmO3
- aesO0
- aesO3
- bfO0
- bfO3
- dfaddO0
- dfaddO3
- dfdivO0
- dfdivO3
- dfmulO0
- dfmulO3
- dfsinO0
- dfsinO3
- gsmO0
- gsmO3
- jpegO0
- jpegO3
- mipsO0
- mipsO3
- mpegO0
- mpegO3
- shaO0
- shaO3
- C-torture
**Coverage Metrics**

**Instruction Coverage (icov)**

\[ icov = \frac{\text{# of checked static operations}}{\text{# of static operations}} \]

**Statement Coverage (scov)**

\[ scov = \frac{\text{# of statements executed at least once at runtime}}{\text{# of static statements}} \]

\textbf{ccov: } C statement coverage — \textbf{vcov: } Verilog statement coverage

**Instruction Coverage ≠ Statement Coverage**

- \textbf{scov} is dynamic while \textbf{icov} is static
- \textbf{icov} has a much finer granularity (operations not statements)
- \textbf{icov} is meant to check how many operations can be checked even if they are not executed
Coverage Results

<table>
<thead>
<tr>
<th>Function</th>
<th>ccov (gcov 4.9)</th>
<th>vcov (Mentor Modelsim SE-64 10.3)</th>
<th>icov (computed by our method)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcmO0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adpcmO3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aesO0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aesO3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfO0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bfO3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfaddO0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfaddO3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfdivO0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfdivO3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfmulO0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfmulO3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfsinO0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfsinO3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gsmO0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gsmO3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jpegO0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jpegO3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mipsO0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mipsO3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpegO0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpegO3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shaO0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shaO3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-torture</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Control Flow Traces (FCT)

Software and Hardware Executions

CDFG and a FSM are typical IRs for HLS compilers.

Consider a CDFG and a FSM for a high-level function.

From a control-flow standpoint, for a given input, they represent SW and HW executions respectively.

**Definition: Software Control Flow Trace (SCFT)**

The SCFT on a given input $I$ is the ordered sequence of BBs representing the execution of the CDFG.

**Definition: Hardware Control Flow Trace (HCFT)**

The HCFT on the same input $I$ is the ordered sequence of states describing the execution of the FSM.
Software and Hardware Operations

Control Flow information is not enough

Cannot spot bugs that do not alter the execution path

A finer granularity is necessary

Definition: Software Op Trace (SOT)

Given a Basic Block $BB_i$ and its associated list of states $S_1, \ldots, S_k$, the Software OpTrace of $BB_i$ is the list of results of the statements in that BB.

Definition: Hardware Control Flow Trace (HOT)

Let $S_j$ be a state in the FSM. The Hardware OpTrace of a state $S_j$ the set of results of the operations scheduled in that state.