POLITECNICO DI MILANO

Debugging and Automated Bug Detection for Hardware Generated with BAMBU

Tutorial @ FPL Conference 2017 – Ghent – Belgium

Pietro Fezzardi

Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
pietro.fezzardi@polimi.it

Outline

- Motivation and Goals
- Background: Hardware Debugging Methodologies
- Automated Bug-Detection with Discrepancy Analysis
- Remarks and Conclusion

Motivation and Goals

Motivation

- What we had
  - Advanced compiler infrastructure
  - Different compiler optimizations and features to test
  - Extensive regression tests for multiple scenarios
  - Co-simulation workflow to identify bugs at the interfaces
- Shortcomings
  - Only find bugs at the interfaces
  - Cannot locate and isolate the first bug
  - Does not automatically backtrack to C code
  - Deep understanding of HLS and the exact optimizations is needed
  - Lots of time and trials-and-errors to do it manually

Ideas and Goals

- Turns out these shortcomings are a general issue
- All HLS frameworks face this challenges in debugging
- Design goals
  - Find bugs with single operation granularity inside the designs
  - Automatically backtrack to C
  - Automatically handle HLS compiler optimizations
  - Automatically handle HW/SW memory mapping
  - Avoid user interaction and manual operations (for regression tests)
  - Do not require to users deep knowledge of HLS engine internals
  - Automatically locate and isolate the first bug

Background

Hardware Debugging Methodologies

The HW Debugging Quadrant

How/When	Online	Offline
On-Chip	?	?
Simulation	?	?

On-Chip vs Simulation-Based

- On-Chip debugging (aka in-circuit debugging)
  - design is synthesized and executed directly on-chip
  - data for debugging is collected by additional tracing circuits
  - the original design must be altered
- RTL simulation
  - design simulated at the RTL level
  - simulation software runs on a host machine

Offline vs Online

- Offline
  - the design is executed completely (or partially until a bug)
  - data for debugging collected during operation
  - the 'real' debugging is performed on the collected data
  - on-chip debugging → tracing circuits are necessary
  - simulation → no modification
- Online
  - execution and debugging are simultaneous
  - stepping/breakpointing/analysis/modification
  - on-chip → controller circuits and external interface are necessary
  - simulation → no modification, but needs simulation support

On-Chip Debugging – Pros

- Extremely fast execution
- Only way to catch HW faults
  - power-supply noises
  - damaged gates
  - interference with the environment
- Only way to catch external communication problems
  - unless the other peer can be interfaced with simulation

On-Chip Debugging – Cons

- Poor controllability and observability
- Additional tracing logic is necessary to:
  - trace and collect signal variations
  - control the device (only for on-line debugging)
  - can compromise timing or ability to replicate bugs
- Necessary trade-off between:
  - area for tracing circuits
  - number of traced signals + time span available for tracing
- Restrictions on memory layouts
- Re-synthesis of tracing circuits to change parameters

Debugging with Simulation – Pros and Cons

Pros:

- Complete controllability and observability
- Designs can be debugged without changing it
- Incremental modifications without regenerating the bitstream

Cons:

- Much longer execution time than the corresponding HW

Automated Bug-Detection

with

Discrepancy Analysis

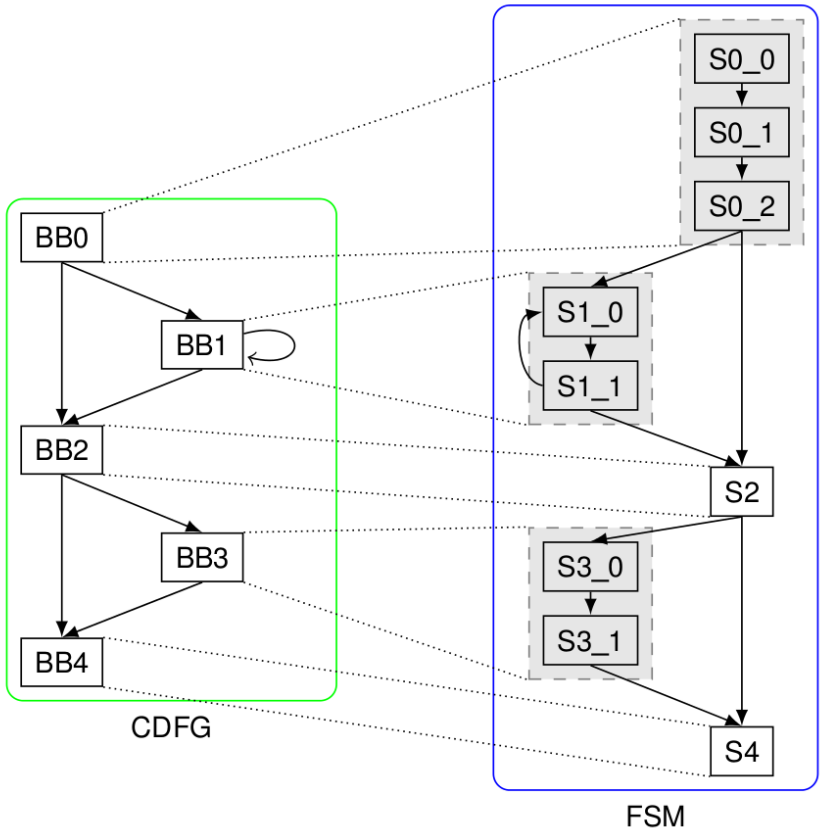
Introduction to Discrepancy Analysis

CDFG → behavior of a function

FSM → generated from CDFG with HLS

CDFG → describes SW execution

FSM → describes HW execution

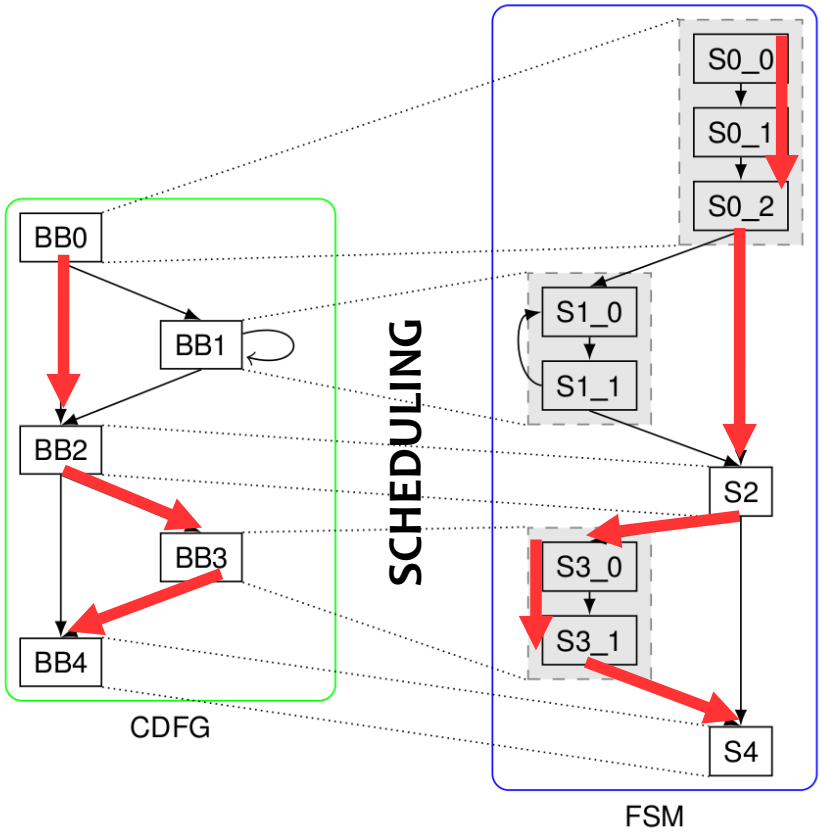


- They are executed with the same inputs
- We want to compare the two executions
- We want to be able to tell if they match
- We do it on 2 levels:
  - **Control flow level**
  - **Operation level**

Control Flow Traces (CFT)

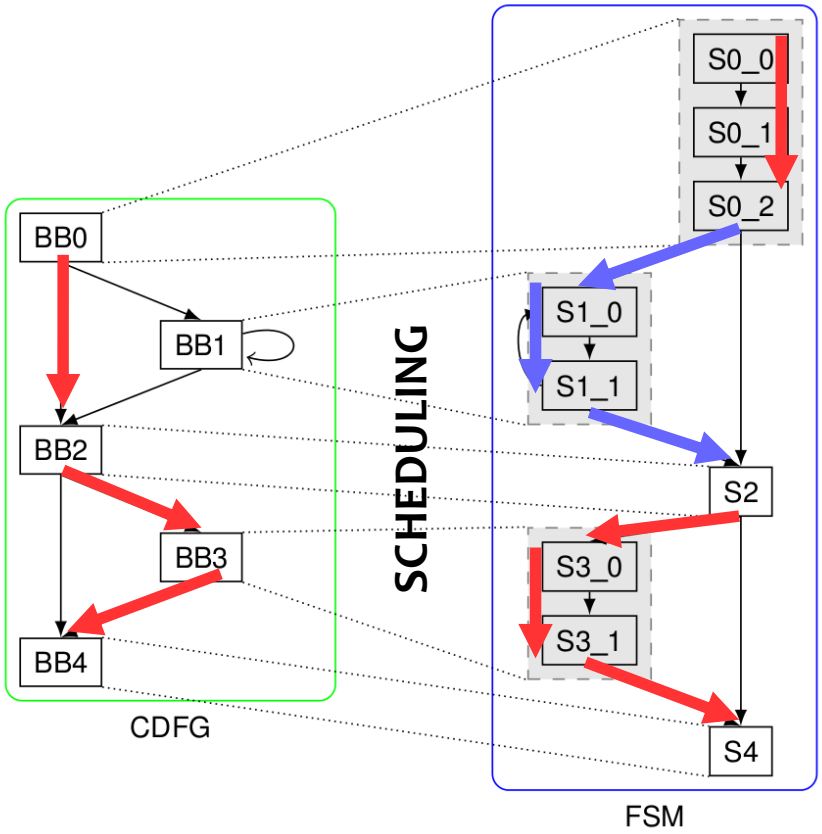
Scheduling maps basic blocks onto states.

- The Software Control Flow Trace (SCFT) is the list of basic blocks traversed by the software execution.
- The Hardware Control Flow Trace (HCFT) is the list of states in the FSM traversed by the hardware execution.

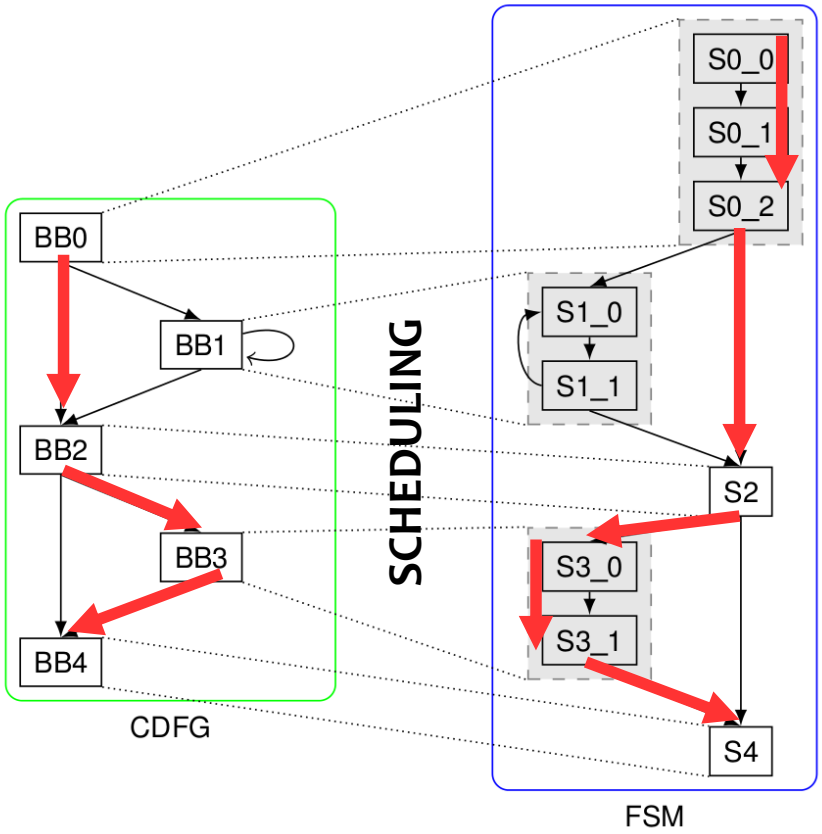


A SCFT and a HCFT are equivalent if the second can be obtained from the first using only scheduling information.

Control Flow Traces (CFT)



NOT EQUIVALENT



EQUIVALENT

Comparing Control Flow Traces

Collecting HCFT:

- Select signals in the controller of every function's FSM
  - start_port
  - done_port
  - present_state
- Dump the value changes during the simulation (VCD)

Collecting SCFT:

- Instrument the high-level source code
- Dump the basic block number at the beginning of every basic block

Comparing CFTs is straightforward using the scheduling map built by HLS

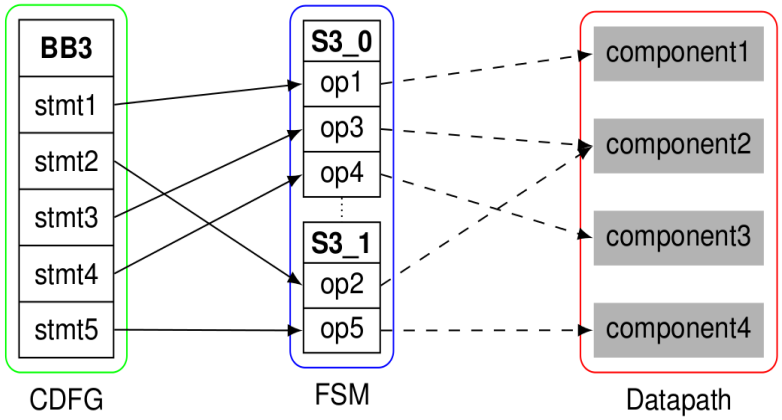
Control Flow Traces - Limitations

- Impossible to find bugs that do not change the control flow
  - wrong values not used for branching cannot be found
- Impossible to determine the moment of first discrepancy
  - the wrong value may alter the control flow long after its creation
- Impossible to isolate the failing component
  - the wrong value can be originated by a component it depends on
- We need a finer granularity → per-operation

OpTraces (OT)

- The Software OpTrace (SOT) of an operation is the list of values assigned to a variable during execution

- The Hardware OpTrace (HOT) of an operation is the list of values of the output signals of the component bounded to the operations, when the FSM is in a state where the operation is scheduled



- A SOT and an HOT are equivalent if all the values of the SSAs assigned in the software operations are equal to the results of the associated operations in the corresponding states of the FSM, modulo some equivalence function.

- Example:
  - SSA assigned in stmt3 == out_signal of component2 when FSM is in state S3_0
  - SSA assigned in stmt2 == out_signal of component2 when FSM is in state S3_1

Comparing OpTraces

Collecting HOTs:

- Select the HW modules corresponding to operations in the FSM state
- It is possible using allocation/binding
- For the selected modules select the out_signal
- Dump the signals during simulation (VCD)

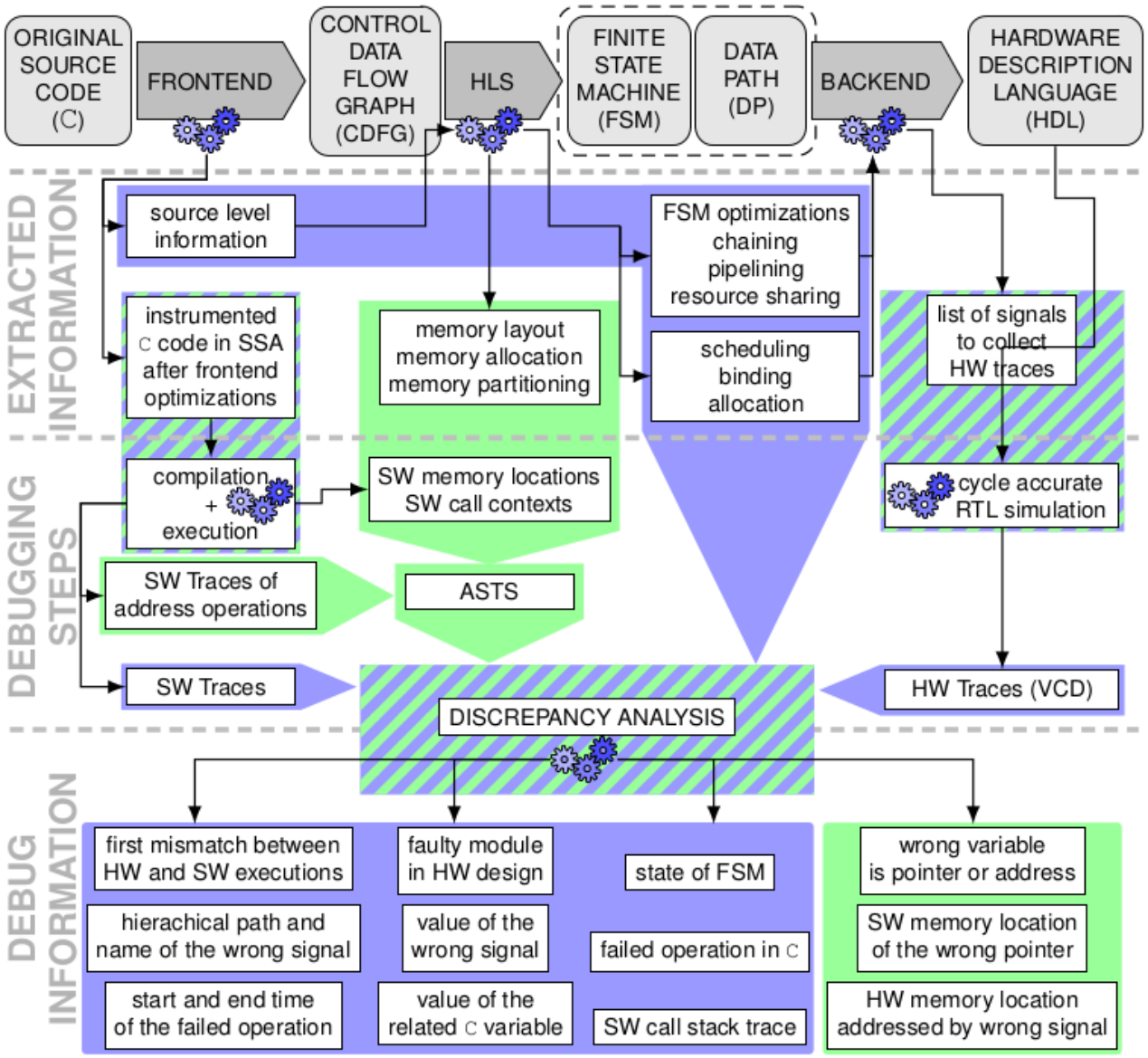
Collecting SOT:

- Instrument the high-level source
- Print values of every SSA variable after assignment

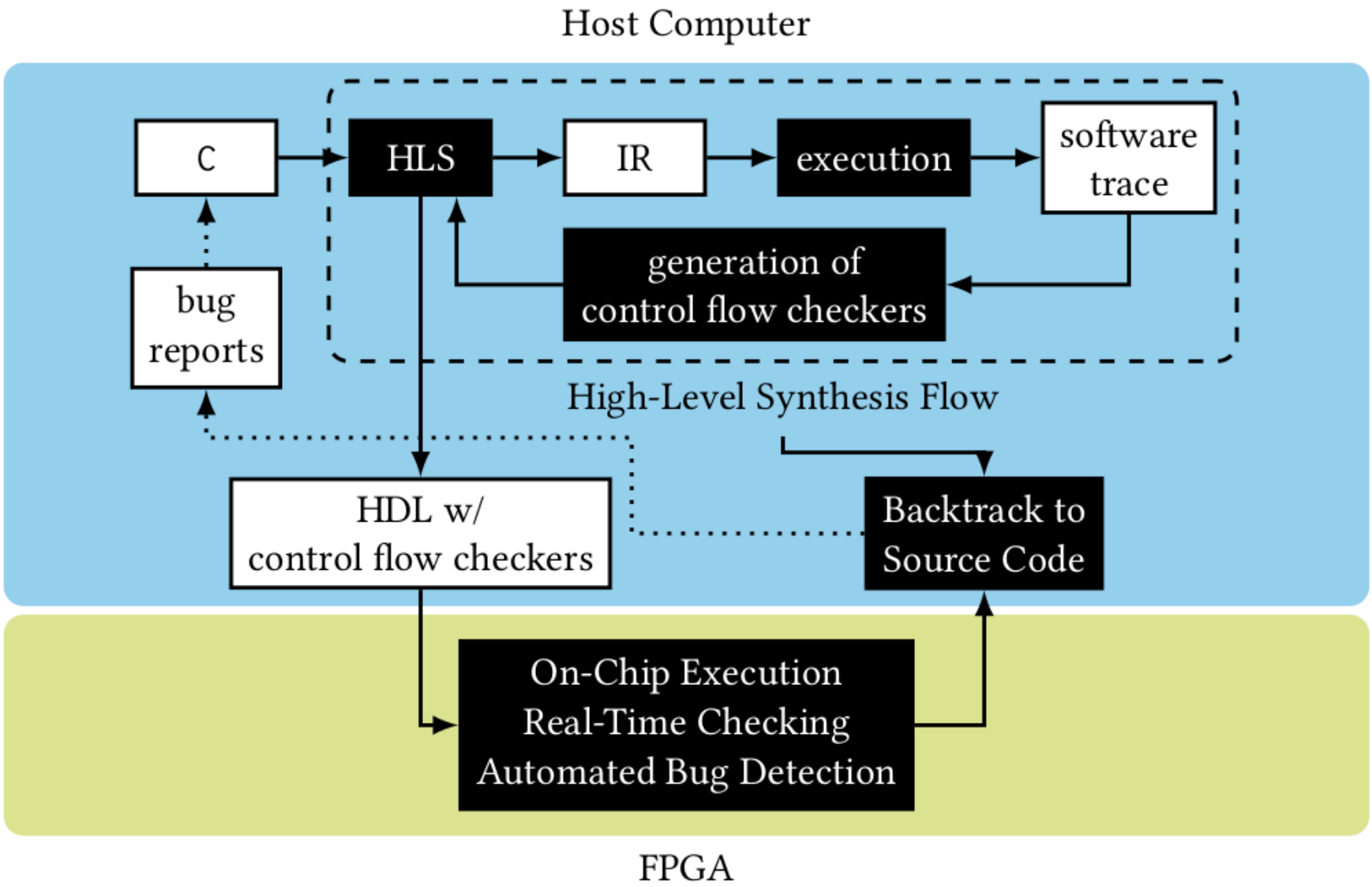
Complexity of comparing the traces is linear with number of assignments

The comparison may be something more complex than bitwise equality
(floating points, pointers, custom data encoding)

Discrepancy Analysis Debug Flow



On-Chip Discrepancy Analysis



Conclusion

Conclusion

Advantages of Discrepancy Analysis

- ✓ Keep relationships between high-level code and HW
- ✓ Automatic selection of the signals necessary for debugging
- ✓ No limits on compiler and HLS optimizations
- ✓ No restriction on memory layouts
- ✓ HW/SW address translation to debug pointers
- ✓ Avoid user interaction user interaction
- ✓ Bug detection with fine-grained per-operation granularity
- ✓ Suitable for use in regression testing
- ✓ Saves lots of time automating error-prone task

Thanks for Your Attention

Questions?

Contacts

emails: pietro.fezzardi@polimi.it

website: https://panda.dei.polimi.it

github: https://github.com/ferrandi/PandA-bambu

BACKUP SLIDES

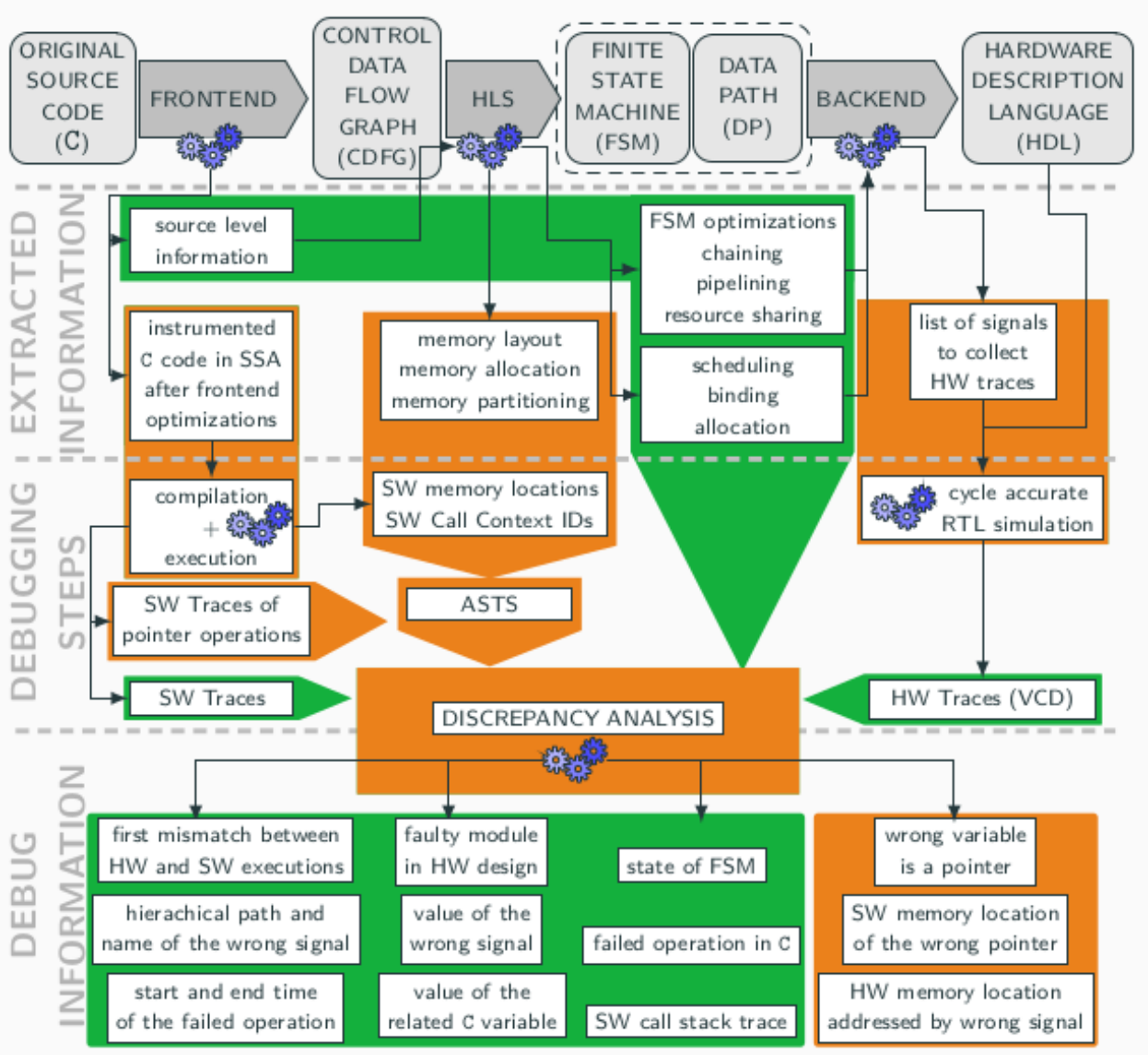
Address Discrepancy Analysis

Handling Pointers and Memory Accesses

Extension to Pointers - Motivation

- OpTrace equivalence is easy for integers
- Pointers in hardware and software are not comparable
- Memory optimizations are critical to performance
- Memory bugs are hard to debug in software
- Memory bugs are harder to find in hardware
- We want the same advantages of Discrepancy Analysis for debugging operations involving pointers

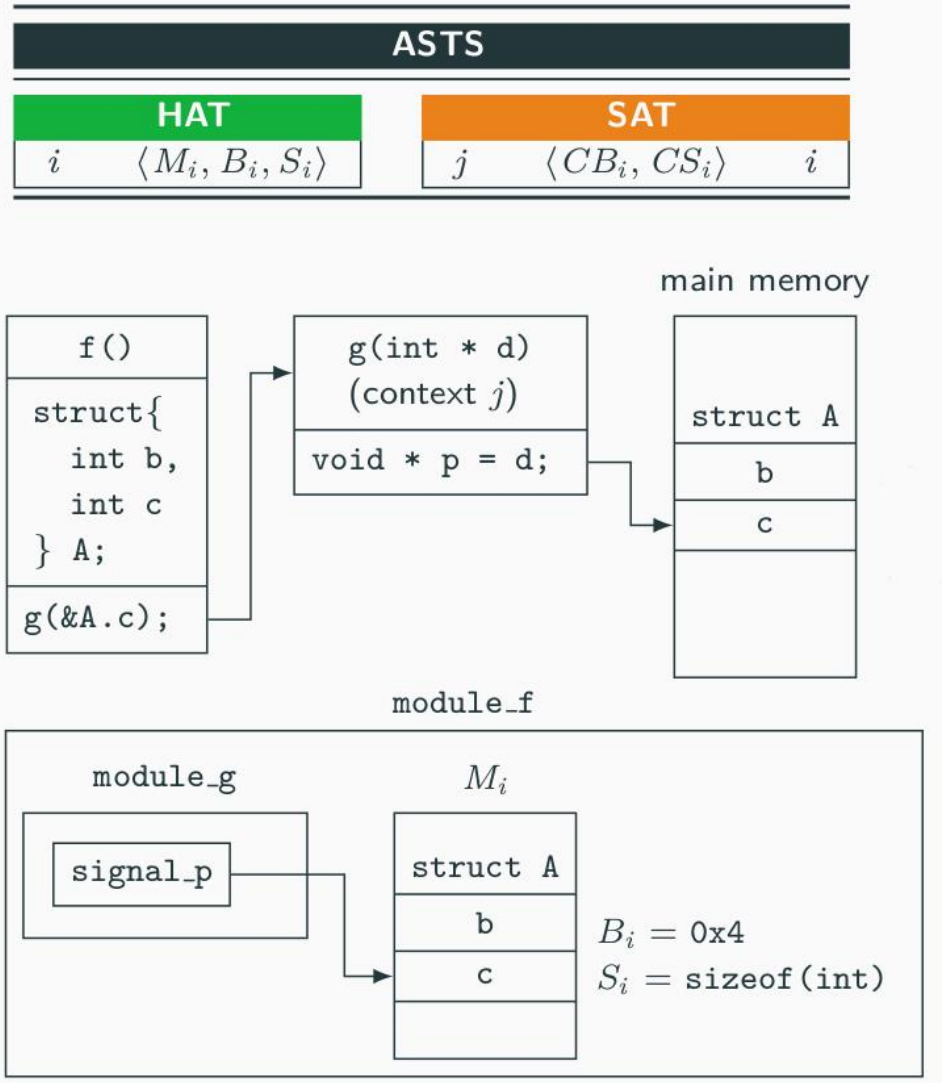
Address Discrepancy Analysis Debug Flow



Address Space Translation Scheme

Shared Data: ASTS = (SAT, HAT)

```
1  bool discrepancy (j, s, h)
       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
2      i = search (j, s) in SAT;
3      if (i is not found) then
4          // s is not in range for any variable
5          return false;
6      else
7          ⟨M_i, B_i, S_i⟩ = search(i) in HAT;
8          if (⟨M_i, B_i, S_i⟩ is not found) then
9              // not memory-mapped in HW
10             return true;
11         else
12             h' = decodeHW (⟨M_i, B_i, S_i⟩);
13             if h ≠ h' then
14                 return true;
15             else
16                 return false;
```



Common Approaches to HW Debugging for HLS

Common approaches to HLS debugging - 1

- Synthesis of ANSI-C assertions
  - offline, both in-circuit and simulation
  - users have to manually insert the assertion in the high-level code
  - cannot spot the place where the bug originates
  - cannot spot bugs not guarded by assertions

Common approaches to HLS debugging - 2

- Trace logic optimization using high-level information
  - on-chip, offline
  - focuses on optimization of the tracing hardware
  - increase observability rather than controllability
  - data collected automatically during operations

Common approaches to HLS debugging - 3

- Software-like debugging
  - online, both with simulation and on-chip (some restrictions)
  - software-like user-friendly features:
    - Stepping
    - Breakpoints
    - Watchpoints
    - Real-Time inspection of the signals
  - good information on high-level source code
  - lots of time wasted stepping and manually inspecting the signals
  - cannot handle optimizations and temporary variables

Shortcomings of the proposed approaches

Challenges

1) relationship (high-level source code ↔ HDL)

2) identify signals for debugging

3) temporary variables and compiler optimizations

4) HLS optimizations

5) independent of memory layouts

6) HW/SW Address Space Translation

- **Assertions**
  - ✓ 4?
  - ✗ 1 – 2 – 3 – 5 – 6
- **Tracing logic optimization**
  - ✓ 1 but not shown to users, recently some support for 3 – 4
  - ✗ 2 – 5 – 6
- **SW -like**
  - ✓ 1
  - ✗ 2 – 3 – 4 – 5 – 6

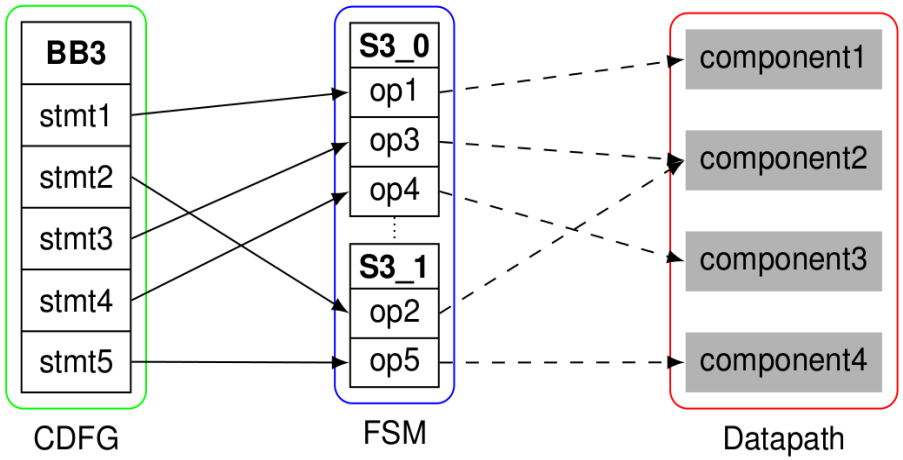
Extras on OpTraces

OpTraces - Preliminaries

Basic Blocks contain statements in SSA form

3 types of statements:

- control flow statements
- SSA assignements
- Φ operations



Control flow is already covered by CFTs

Φ operations can be converted in sets of assignments

We only need to handle assignments