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

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

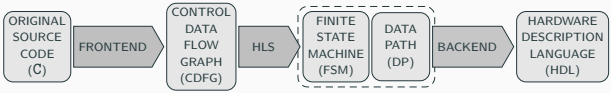
General Ideas

- 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

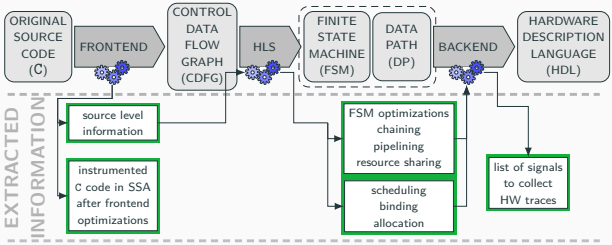
Goals Related to Pointers

- 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

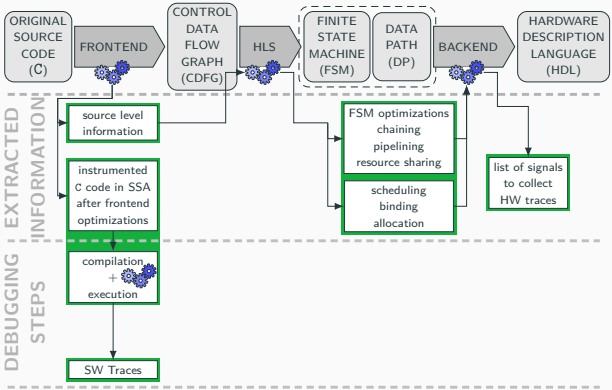
Discrepancy Analysis Debug Flow



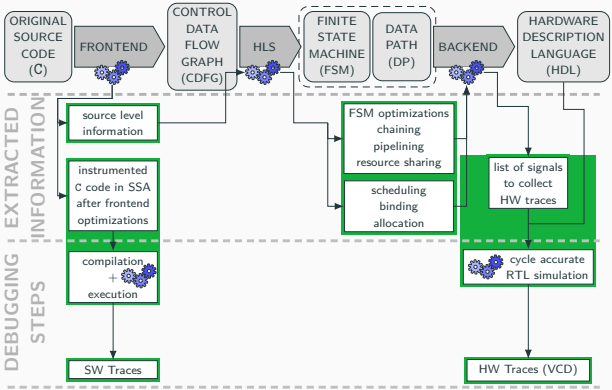
Discrepancy Analysis Debug Flow



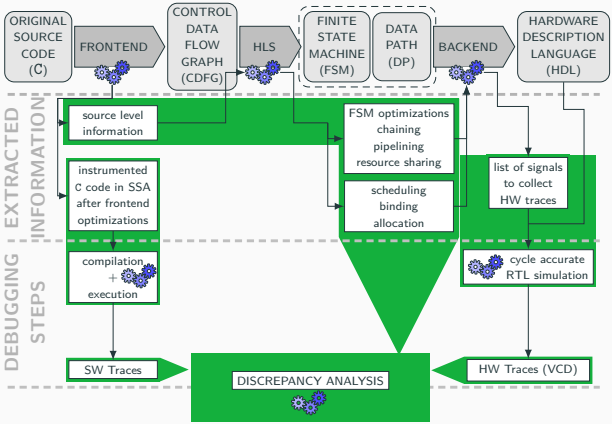
Discrepancy Analysis Debug Flow



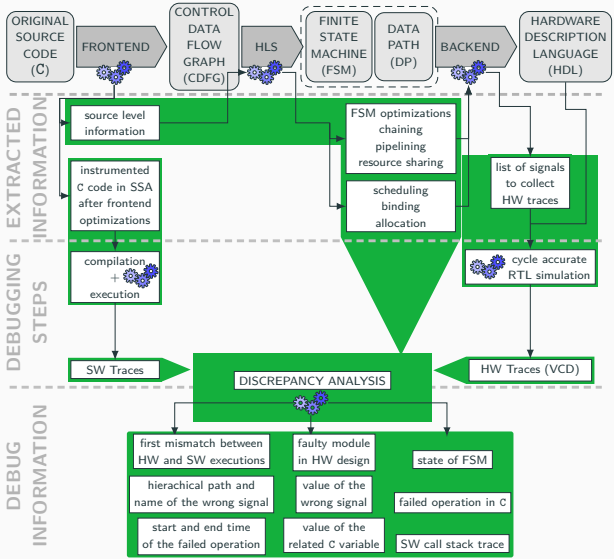
Discrepancy Analysis Debug Flow



Discrepancy Analysis Debug Flow



Discrepancy Analysis Debug Flow



Memory Locations

Memory Location

A **Memory Location** $\langle M_i, B_i, S_i \rangle$ is an unambiguous representation of a position in memory

In HW	In SW
◦ M_i: unique identifier for a memory module	◦ M_i: can be omitted
◦ B_i: an offset in the memory module identified by M_i	◦ B_i: address in main memory
◦ S_i: size of the memory location	◦ S_i: size of the memory location

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

Memory Allocation

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

Address Space Translation Scheme (ASTS)

Hardware Address Table (HAT)	Software Address Table (SAT)
$i \quad \langle M_i, B_i, S_i \rangle$	$j \quad \langle CB_i, CS_i \rangle \quad i$

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

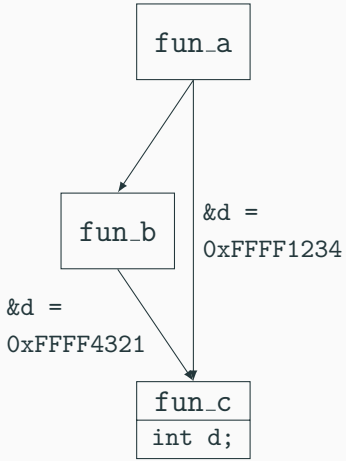
Different Memory Location at every function call

An ID is necessary to distinguish between calls

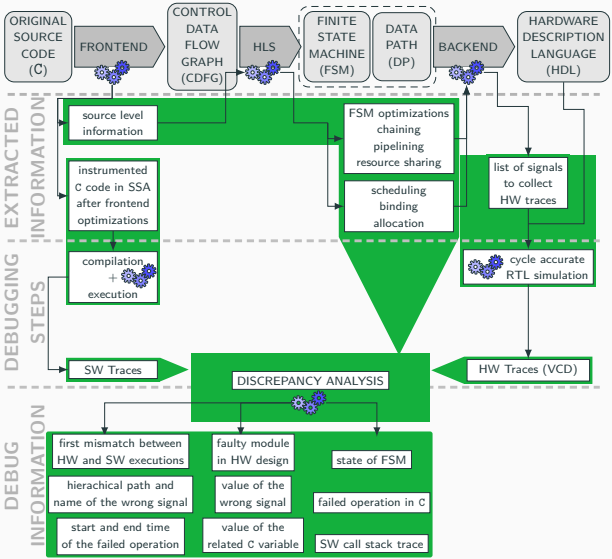
An ID uniquely identifies a path on the call graph

Function calls are instrumented in the `C` code

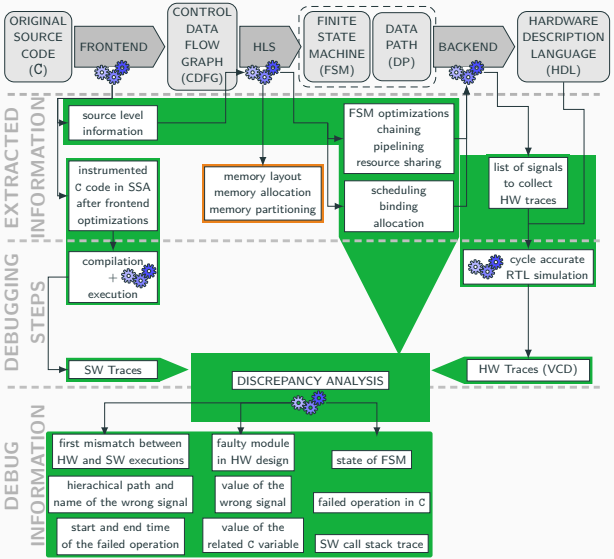
Context ID and memory mapping are printed at runtime



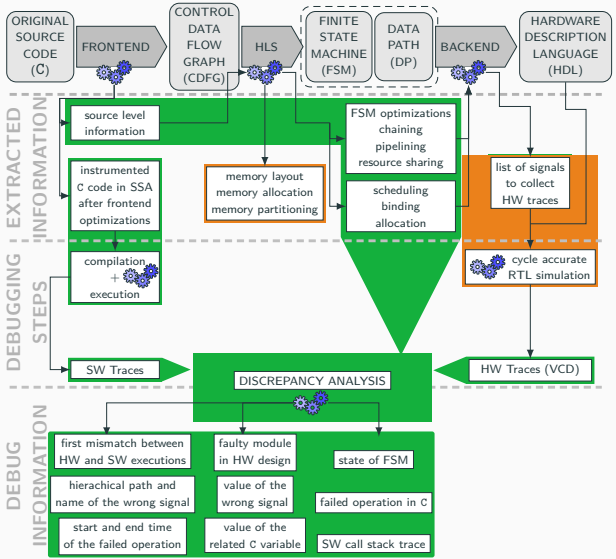
The Extended Address Discrepancy Analysis Flow



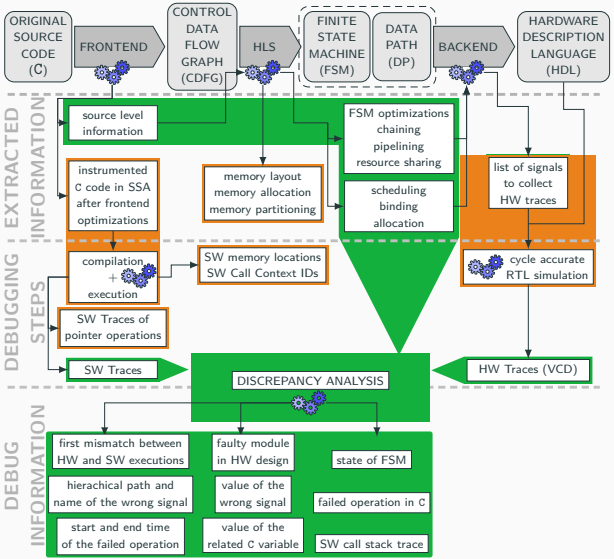
The Extended Address Discrepancy Analysis Flow



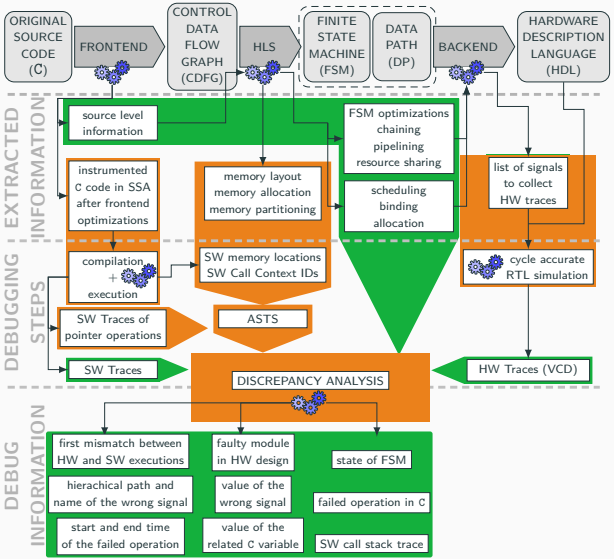
The Extended Address Discrepancy Analysis Flow



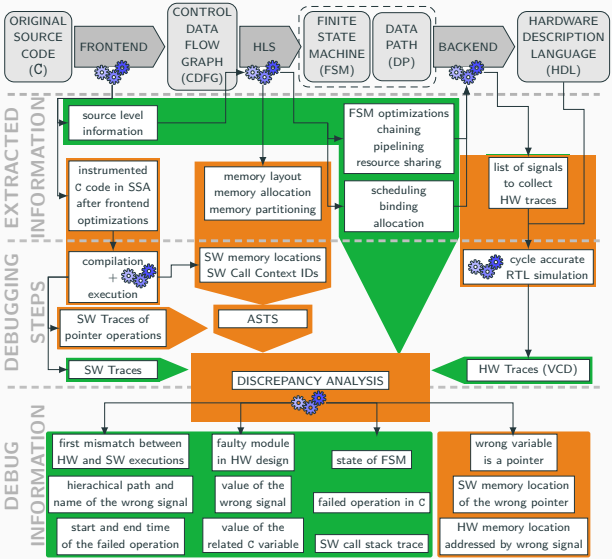
The Extended Address Discrepancy Analysis Flow



The Extended Address Discrepancy Analysis Flow



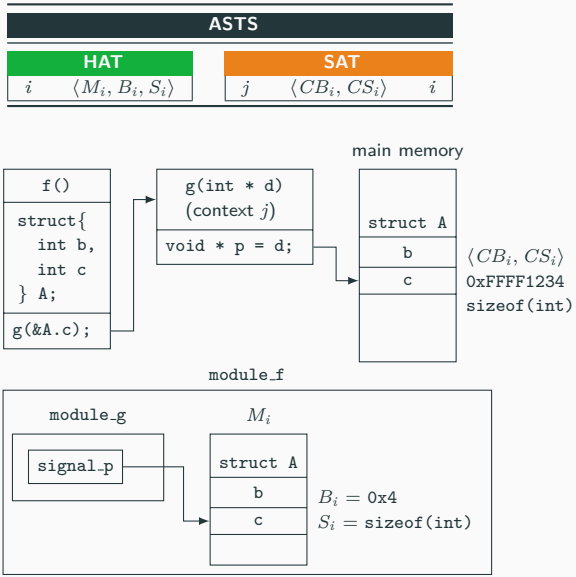
The Extended Address Discrepancy Analysis Flow



Address Discrepancy Algorithm

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;
```



Evaluated HLS Tools

Bambu [Pilato et al; CODES+ISSS'11]

- Developed at Politecnico di Milano
- Based on GCC (4.5 up to 6)
- Free Software (GPLv3)

Commercial Tool

- Production-ready
- Recent version (late 2015 - early 2016)
- Targets Xilinx FPGAs
- Closed source proprietary license

LegUp [Canis at al.; TECS'13]

- Developed at University of Toronto
- Based on LLVM
- Free for non-commercial not-for-profit use

Benchmarks

CHStone [Hara et al.; ISCAS'08]

- Well known benchmark suite for HLS
- Both contol- and data-oriented examples
- 12 self-contained `C` programs
- Try to settle a common ground for HLS tools

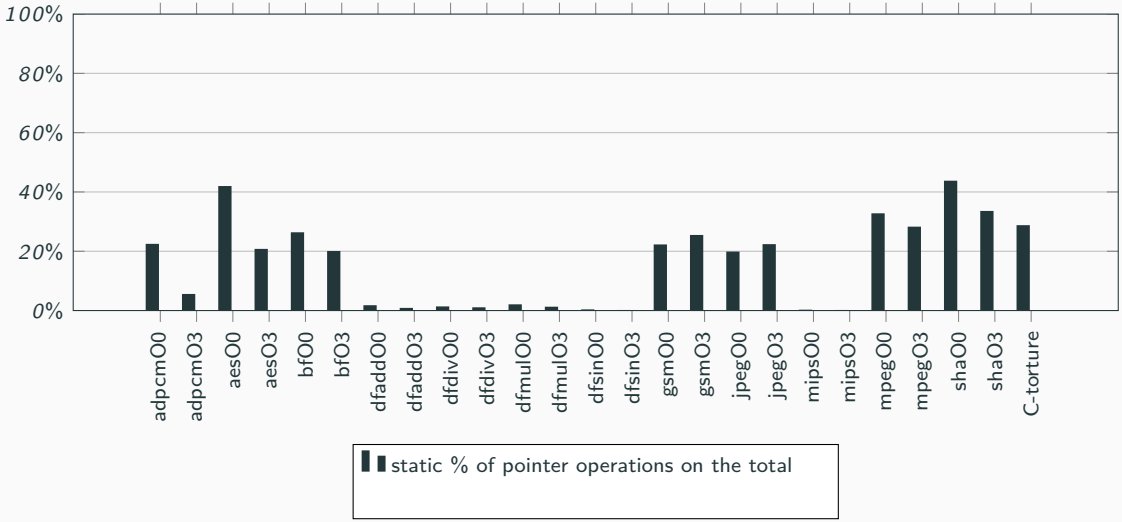
GCC C-torture

- 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

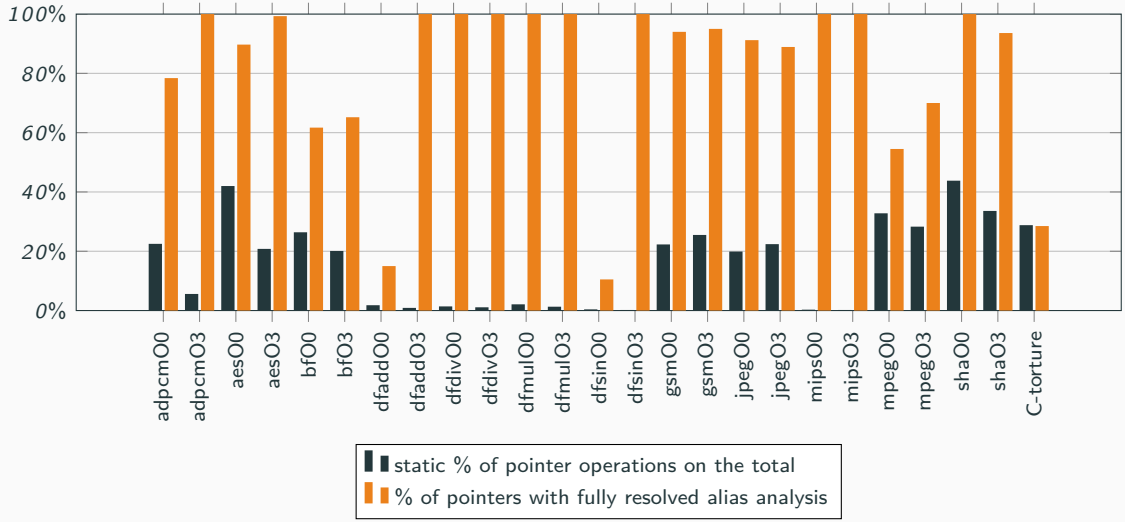
Test Matrix

	CHStone	GCC C-torture
Bambu	✓ fully automated	✓ fully automated (several bugs found)
Commercial Tool	use partitioning directives manual bug insertion imitate known bugs found in Bambu manual reconstruction of ASTS manual execution	56/216 failed HLS manual on a short list
LegUp	✓ partially automated	✓ partially automated (on a short list)

Significance of Pointer Operations



Significance of Pointer Operations



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

Example where Commercial Tool fails HLS

```
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;
}
```

False Positive

```
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

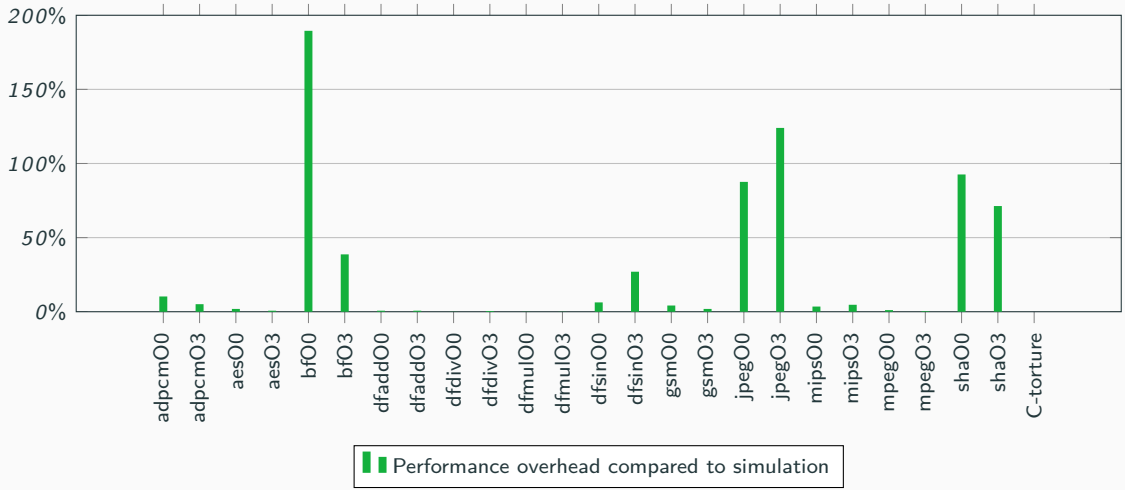
Address Discrepancy Analysis do not check out-of-bound addresses

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



Coverage Metrics

Instruction Coverage (icov)

$$\textbf{icov} = \frac{\#\text{ of checked static operations}}{\#\text{ of static operations}}$$

Statement Coverage (scov)

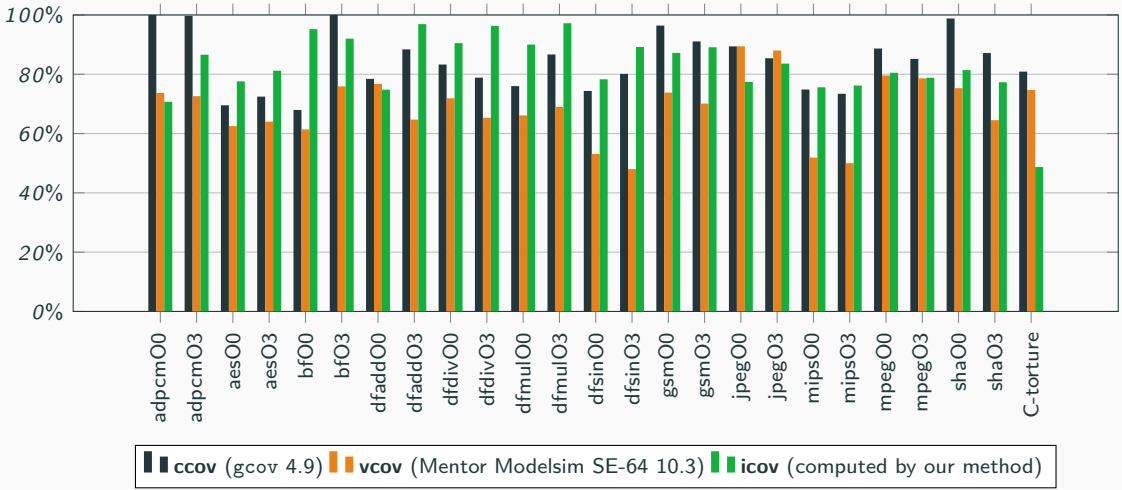
$$\textbf{scov} = \frac{\#\text{ of statements executed at least once at runtime}}{\#\text{ of static statements}}$$

ccov: `C` statement coverage — **vcov**: Verilog statement coverage

Instruction Coverage $\neq$ Statement Coverage

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

Coverage Results



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

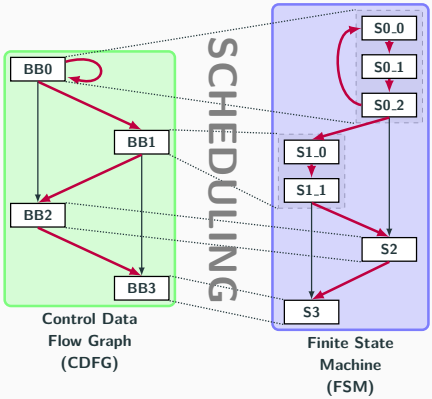
From a control-flow standpoint, for a given input, they represent SW and HW executions respectively

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 the ordered sequence of states describing the execution of the FSM



OpTraces (OT)

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.