Trace–Based Automated Logical Debugging for High–Level Synthesis Generated Circuits

Pietro Fezzardi
Politecnico di Milano, Milano, Italy
pietro.fezzardi@polimi.it

Michele Castellana
Politecnico di Milano, Milano, Italy
michele.castellana@mail.polimi.it

Fabrizio Ferrandi
Politecnico di Milano, Milano, Italy
fabrizio.ferrandi@polimi.it

***Abstract*—In this paper we present an approach for debugging hardware designs generated by High–Level Synthesis (HLS), relieving users from the burden of identifying the signals to trace and from the error–prone task of manually checking the traces. The necessary steps are performed after HLS, independently of it and without affecting the synthesized design. For this reason our methodology should be easily adaptable to any HLS tools.**

The proposed approach makes full use of HLS compile time informations. The executions of the simulated design and the original C program can be compared, checking if there are discrepancies between values of C variables and signals in the design. The detection is completely automated, that is, it does not need any input but the program itself and the user does not have to know anything about the overall compilation process. The design can be validated on a given set of test cases and the discrepancies are detected by the tool. Relationships between the original high–level source code and the generated HDL are kept by the compiler and shown to the user. The granularity of such discrepancy analysis is per–operation and it includes the temporary variables inserted by the compiler. As a consequence the design can be debugged as is, with no restrictions on optimizations available during HLS.

We show how this methodology can be used to identify different kind of bugs: 1) introduced by the HLS tool used for the synthesis; 2) introduced using buggy libraries of hardware components for HLS; 3) undefined behavior bugs in the original high–level source code.

I. Introduction

High–Level Synthesis (HLS) aims to increase the productivity of hardware designers, using high–level programming languages to overcome the challenges of an always increasing design complexity. However there are still obstacles to achieve this goal. Among them there are the limitations of hardware (HW) debugging techniques and tools specifically tailored for HLS. For this reason in recent years there have been multiple efforts to remove such limitations. Indeed some of the proposed solution are big steps forwards, but they are based on assumptions and requirements that considerably limit the available options when performing HLS. Typical requirements are to avoid function inlining and local RAMs. Another peculiar trait of all the debugging tools for HLS–generated circuits is the poor accuracy in case of heavy compiler optimizations. This is due to the fact that allowing these options severely affects the observability of signals.

In this paper we describe an approach for automatic detection of logical bugs in HW designs generated using HLS. Our main goal is to remove the restrictions described above, allowing the usage of all the available options for the memory layouts and all the compiler optimizations when performing HLS, without compromising the possibility to debug the generated circuit without modifications. All of this preserving the ability to relate the generated HDL to the original high–level source code. These goals are very important because we do not want to limit our debugging methodology to unoptimized circuits, and we want to be able to spot bugs introduced by any optimization step performed in HLS. We describe our implementation of a debugging tool to show a real–world application of the described methodology. The debugger is designed as a set of passes for the open source bambu HLS compiler [1], which performs HLS from C.

The remainder of this work is organized ad follows: section II describes the specific problems of debugging HW designs, firstly from a general standpoint and then focusing on HLS. Section III formalizes our approach for our automated bug detection. Section IV describes how it is practically used in our implementation. Section V describes the results in terms of detected bugs, performance and coverage. It shows different potential scenarios for effective use of the methodology. Finally section VI summarizes the results, discussing some cases still not covered by the debugger and outlining a possible evolution of the research.

II. Background and Related Work

A. Approaches and challenges in hardware debugging

Debugging HW designs is a complex process. Typically it involves selecting a large number of signals, tracing their values concurrently during the execution, and analyzing them to find misbehaviors. To effectively do this, any HW debugging technique needs to provide three main features [2]:

1) *signal observability*;
2) *hardware controllability*;
3) *limited turnaround times*.

Signal observability is the ability to see the values of the largest number of signals in the design, with the finest granularity, across the largest time span as possible. *Hardware controllability* is fine control on the design execution during the debug operation. It is necessary to detect not only the wrong signal, but also the exact time when that happens and possibly the values of a number of other signals in a surrounding time frame. By *limited turnaround time*, instead,

we mean that the time needed between a request to the debugger and the attainment of the result must be short enough not to slow down the whole development process. Achieving these goals usually requires some trade off. Research efforts aimed at maximizing the performance in these fields can be subdivided mainly in two groups:

- approaches for debugging the circuit directly on–chip;
- debugging techniques based on RTL simulations.

Both the methodologies have intrinsic characteristics that makes particularly easy to provide some of the mentioned features rather than others.

On one hand, debugging on–chip is the only way to spot malfunctions due to HW faults (power–supply noises, environmental interferences, damaged gates). However, for logical bug detection, in–circuit debug is usually worse than simulation in providing the three features described above. Indeed, to guarantee observability and controllability, it is necessary to add tracing circuitry or to enforce restrictions on the memory layout of the HW accelerators. This does not scale with increasing design complexity. It also imposes limits on the number of traced signals and on the time frame that can be captured. The extra logic or the forced memory layout may modify the original design, compromising crucial timing characteristics of the accelerator, or making the bug non–reproducible. Even when the insertion of the debugging circuits is harmless, finding and analyzing the interesting signals can be a hard task. Another problem of in–circuit debugging is related to the turnaround time. The place and route process is slow and development can get severely hampered by the need to re–synthesize the bitstream whenever the tracing logic has to be changed. To avoid this problem, many studies showed different approaches for post–synthesis insertion of debugging circuits, like [3], [4] and [5], but some reconfiguration of the bitstream is always necessary.

On the other hand, debugging at the RTL level using simulation is far slower, but this do not inevitably leads to longer turnaround times. Indeed simulation takes much more time than real HW execution, but incremental modifications can be done during the debug process without altering the bitstream. In addition, no extra circuits or dedicated memories are needed to provide signal observability, since in software (SW) memory limits are not an issue anymore. Controllability can be guaranteed using simulators' APIs, enabling to execute the design until a determined point, stop it, analyze variables and then resume the execution (or even roll it back) without affecting the logic of the simulated circuits. If the values of the signals are dumped to file, using the standard Value Change Dump format [6], there is even no need for controllability at all, because the HW can be simulated and the values inspected later. In this way simulation succeeds in achieving complete observability of the signals. However the difficulty in finding the interesting ones remains. Even if all the variations of every signal for the entire execution are saved in VCD during the simulation, the programmer has to inspect manually the registered traces. For HLS–generate circuits there is an additional degree of complexity, due to the problem of determining the relationship between the generated HDL description and the high–level source code. This complexity grows considerably with compiler optimizations and it is also dependent on several modifications introduced by the HLS engine.

B. Recent advances in debugging HLS–generated circuits

In the recent years many results have been pushing the limits of debug capabilities for electronic circuit designs generated by HLS tools. The idea they all have in common is to take advantage of the extra informations present in the original high–level specification and use it for debug purposes, leveraging the patterns in the generated HW due to the predictability of the HLS engine. In addition to what said in section II-A, debugging methodologies targeted to HLS–generated designs can be roughly grouped in two families: optimizations of tracing circuits for in–system debug and methods to bring SW debug look and feel to HW verification.

In particular, the researches centered on debugging on–chip mainly focus on increasing controllability and observability, using high–level knowledge to fine–tune the trace buffers. [7], for example, shows a technique for maximizing the observable events with low area overhead. The tracing circuits described by the authors can be inserted with small modifications and they can be optimized using high–level knowledge of the Control Flow Graphs and State Transition Graphs used for HLS. However they do not scale very well with the number of the signals. Moreover, the paper purposely skip over the problems of the individuation of the interesting signals and how to relate them back to the high–level original source code. The authors just say they assume it can be done using compiler informations. In addition, given that the trace buffers are per–signal, additional processing is required to reconstruct the time relationships between the traces. This becomes very hard when the necessary variables are not stored in a register, or even impossible when heavy compiler optimizations are activated. Another trend in on–chip debugging is based on bringing ANSI–C assertions to HW [8] [9] [10]. This is done adding assertion checker circuits. The checker's Finite State Machine (FSM) can be executed concurrently to the controlled module ([8] and [9]) or the synchronization can be directly performed by the FSM of the accelerator itself (see [10]). Besides area overhead and modifications to the FSMs the main problem of such approach is that it can only check malfunctions foreseen by the developers. The assertion must be manually inserted in the original C specification. This fails to spot bugs that are not checked with assertions. Even when the assertion checker spots a wrong condition, the fault where it comes from can be caused by a previous bug which is difficult to find. Finally, if the buggy HW happens to enter in a hanging state, the relevant assertion trigger point may be never reached at all.

As we can see, all the mentioned results are really focused on efficient implementation of HW tracing circuits to improve the collection and reconstruction of signal traces. But what they leave aside is another set of complex problems. In particular, they still leave to the designer the burden of finding

out the interesting signals and to step through the execution to find bugs. These tasks can be overwhelming with increasingly larger designs. With complex designs it can take days and things are even more complicated if the circuit description is generated by an HLS tools, because the developer has no knowledge of the signal naming conventions and how the signals are related to high–level variables.

The authors of [11], [12] and [13] try to tackle these limitations, bringing to HLS some of the typical software–like debugging methodologies: stepping, breakpointing and dynamic variable inspection. While [11] does not describe an actual implementation, [12] provides such features using dedicated control circuits and trace buffers. These circuits enable the user to control the clock, the execution of the design and to analyze the memory during the operation of the HW accelerator on–chip. The limit is that the design cannot be executed at full speed and the memory layout is constrained to be a single central memory. The only variables that can be analyzed are those stored in the global memory, making temporary variables inserted by the synthesizing compiler invisible to the debugger. The approach and the goals of [13] is quite similar, but the work is more focused on how to keep source–level information than on how to provide observability and controllability of the HW. Notwithstanding the interest towards high–level source informations, the temporaries introduced by compiler optimizations cannot be inspected and several other constraints are imposed to HLS. Namely, inlining is disabled, local RAMs are made global, and constants cannot be stored on ROMs. Interestingly [13] is possibly the first work where the idea of an automated discrepancy detection is outlined, for both on–chip and simulated debugging workflows.

III. Automated Trace–Based Bug Detection for Circuits Generated by High–Level Synthesis

A. Fundamental ideas and goals

The idea of an automated discrepancy analysis is extremely powerful, because it is the best way to guarantee short turnaround times. Controllability and observability are useless if users waste lots of time finding out which are the signals to trace and performing manual stepping to find bugs. Watchpoints and breakpoints are helpful, but they require to spot the right place to insert them and this is not always possible with heavy compiler optimizations. Bringing a software–like debug flow to the HW basically throws away all the performance boost coming from debugging on–chip, because the largest amount of time is spent resynthesizing the tracing circuits and stepping the design. At the same time it imposes limits to optimizations and prevents the design to be tested in the exact form where the bug is found. This is unrealistic, because some bugs could be caused exactly by some front–end optimizations, HLS optimizations or memory layouts.

For these reasons automated bug detection can be the key to really make the design/debug cycles more productive. This approach can be even more useful if it is not dependent on particular memory layout and if the granularity of the checks can capture bugs related to temporaries inserted by compiler optimizations. In this respect proposed debug methodology tries to achieve the following goals:

1) perform fast, scalable and reliable automated discrepancy analysis, to detect if and where the original high–level description differs from the HW behavior;
2) avoid any direct interaction with the user during such analysis, to avoid slowing down the process;
3) use HLS informations to provide per–operation and per–variable granularity to the discrepancy analysis, independently of the optimization applied by the compiler and even when chaining and pipelining are enabled;
4) select all the necessary signals automatically; this is fundamental to make the most of the HLS informations, so that the user does not need to know anything about the HLS compiler internals;
5) keep informations on the relationships between the original C code and the generated HDL and show them to the user with useful details if a discrepancy is found;

Clearly points 3 and 4 cannot be achieved in practice without relying on the internals of the HLS framework used for the implementation, but the approach we are going to describe is generic enough to be applied to any HLS tool. These same two requirements impose to have the maximum observability of the signals. This is the major reason of the decision to implement our proof–of–concept relying on simulation. However the key contributes of the research (i. e. the automated signal selection and the automated bug detection with maximum granularity) are not strictly dependent on the simulation flow. If the observability of the necessary signal can be achieved in HW, then the same approach should be applicable to in–circuit debug. This should remove the biggest bottleneck to the speed of our method, which is currently the poor performance of RTL simulator compared to HW execution. In this paper we rely on simulation for signal tracing, using different fine–tuned optimizations to improve scalability with the size of the design and the number of execution cycles. Details are given in the following sections. Results of the debugging and performance of the algorithms are shown in section V.

Before diving into the definitions there is an aspect to underline. Even if at first sight our approach can resemble equivalence checking, we are not trying to guarantee formal equivalence between the high-level source code and the generated HW. Rather, for a given input set, we want to extend the granularity of functional verification to find bugs at every level in the HW hierarchy. The goal is to identify the exact time of malfunctions and to isolate the faulty operation/component. In this respect, the original C source code represents the original specification and the generated HW is the design to be tested for equivalence. This allows to validate the HLS engine as well. The discrepancy analysis we are going to introduce is not enough to guarantee formal equivalence, because it works on a input test set. What it does, instead, is, given an input triggering a bug, is to detect the misbehavior automatically, selecting all the necessary signals to reach per–operation gran-

ularity and providing useful information on the location and the cause. We did not investigate how to generate test inputs to provide complete function coverage and branch coverage. This is actually an orthogonal problem per se, especially because there is no straightforward relationship between coverage in C and in Verilog. The topic is extremely vast and it would deserve a separate analysis.

B. Formal description of the problem

We now introduce *Hardware Traces* (HT) and *Software Traces* (ST), describing the execution of HW and SW on a given input. We alse define when they are equivalent. For the definitions we start from the Control Data Flow Graph (CDFG), the FSM and the Datapath (DP), which are general internal representation used in HLS compilers [14]. The CDFG is built by the compiler front–end and it represents the behavior of the original program, with control dependencies between basic blocks (BB) and data dependencies between operations in the BBs. It is also the data structure manipulated by several front–end optimizations, like speculation, code motion, dead code elimination, constant propagation and others. The FSM and the DP describe the synthesized HW. They are created by the compiler during HLS, based on the resulting CDFG after front–end optimizations. This conversion typically requires three tightly related steps: scheduling, allocation and binding. The whole process can also involve non–trivial modifications, like sharing, chaining, pipelining and duplication of operations in more than one state. For a given CDFG, the scheduling tells, for every operation in a BB, the state of the derived FSM where it will be executed. What is important for our purposes is that the HLS engine creates the FSM from the CDFG in such a way that every BB is mapped onto a chain of states in the FSM, where the only control flow instructions are in the last state of the chain (see Fig. 1). This is not strictly true if the FSM implementation used in HLS tool is based on guard conditions [14], but the following definitions can be adapted to this case as well.

1) Control Flow Traces (CFT): Consider a function `f` described in a high–level language such as C, and its CDFG after front–end compiler optimizations. From such a CDFG, HLS produces a FSM and a DP. With the appropriate conventions the two representations can accept the same inputs. For a given input, the CDFG represents the execution of the software (as optimized by the compiler) and the FSM and the DP together the execution of the generated HW. The two flows have different semantics for operations. In BBs they are sequential, while all the operations in a state of the FSM are executed concurrently (or at least in chaining). However, from a control flow standpoint, for a given input I the execution can be described as an ordered list of nodes visited on the graph, being it BBs for CDFG or states in FSM.

Definition 1. Consider a CDFG and an FSM coming from the same high–level specification. We call *Software Control Flow Trace* (SCFT) on a given input I the ordered sequence of BBs representing the execution of the CDFG on the given input.

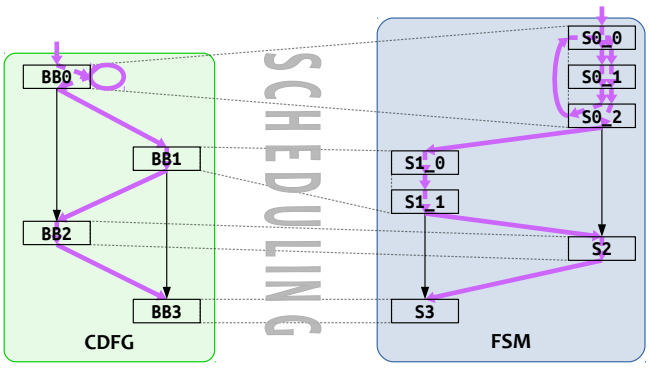

Fig. 1. Scheduling relationship between CDFG and FSM. The thick magenta arrows represent the Control Flow Traces.

We call *Hardware Control Flow Trace* (HCFT) on the same input I the ordered sequence of states describing the execution of the FSM. We will refer to SCFTs and HCFTs together with the name *Control Flow Traces* (CFT).

According to this definition can we see the CDFG as a function S_{CF} that associates a Software Control Flow Trace $S_{CF}(I)$ to every input I. In Fig. 1 the SCFT is (BB0, BB0, BB1, BB2, BB3). In the same way we can regard the FSM as a function H_{CF} that associates an Hardware Control Flow Trace to every input I. In Fig. 1 the HCFT is (S0_0, S0_1, S0_2, S0_0, S0_1, S0_2, S1_0, S1_1, S2, S3).

Definition 2. Let now be fixed an input I for both a CDFG and its associated FSM. Let then be $S_{CF}(I) = \langle BB_0, BB_{k1}, BB_{k2}, \ldots, BB_{K(I)} \rangle$ the related SCFT and $H_{CF}(I) = \langle S_0, S_{j1}, S_{j2}, \ldots, S_{J(I)} \rangle$ the related HCFT. We say that $S_{CF}(I)$ is *equivalent* to $H_{CF}(I)$ if $H_{CF}(I)$ can be produced from $S_{CF}(I)$ simply substituting (BB_k) with the corresponding states through the scheduling relationship. In the following pages we will sometimes say that $S_{CF}(I)$ and $H_{CF}(I)$ *match*. It is possible to show that for every SCFT s there is one and only one equivalent HCFT h.

2) OpTraces (OT): Until now we defined equivalence between HW and SW executions at the control flow level. But our goal is to compare the traces at the operation level. With CFTs we can tell if two executions are not equivalent but not the precise operation causing the mismatch. To reach this granularity we have to consider binding and allocation in addition to scheduling. Fig. 2 shows how the list of statements in a BB can be reordered and assigned to operations scheduled in different states of the FSM. The dashed arrows on the right represent how the operations are bounded to allocated HW components in the DP. Note the mapping of operations on HW components is many–to-one, meaning that components can be shared by multiple operations. Instead, there is a one–to–one mapping between the statements in a BB and all operations scheduled in the related states. Again, this is not strictly true in case of FSMs with guard conditions or duplicated operations, but the described approach can be adapted with slight modification to support them. In this paper we describe only

the basic case for the sake of conciseness. The fundamental point is that every statement cannot be scheduled twice in a chain of states representing a BB. For this reason, even if the semantic of the operations is sequential in BBs and concurrent in states, we can define equivalence also at the operation level.

Definition 3. Given a Basic Block BB_i and its associated list of states $S_1, \ldots, S_k$, we call *Software OpTrace* (SOT) of BB_i the list of results of the statements in the basic block and we denote it with $S_O(BB_i)$. We call *Hardware OpTrace* (HOT) of a state S_j the set of results of the operations scheduled in S_j and we denote it with $H_O(S_j)$. These results are actually the values of the output signals of the HW component implementing the operations themselves, when the FSM is in the state S_j. Note that there is a one–to–one relationship between statements in $S_O(BB_i)$ and all the operations in $H_O(S_1), \ldots H_0(S_k)$. We will use the term *OpTraces* (OT) to denote SOTs and HOTs together.

Definition 4. We say that the OpTraces $S_O(BB_i)$ and $\langle H_O(S_1), \ldots H_O(S_k)\rangle$ are *equivalent*, or that they *match*, if the results of the statements in BB_i are equal to the results of the associated operations in $S_1, \ldots, S_k$.

As stated above, more than one operation can be mapped on the same HW module in the DP. This represents a challenge in detecting the correct value for the result of a given operation. However if a component is shared between multiple operations, they must be scheduled in different states, so it is enough to pick the output value of the component when the FSM is in the correct state to retrieve the correct result for the operation. It is possible to handle operations in chaining, pipelined modules and multi–cycle operations. The only requirement is to know their ending time, but this is easily done given that the execution time of an operation is known to HLS tools. The HCFT allows to identify the time when an operation is started, then the execution time tells when the result must be checked. One exception could be represented by function calls and unbounded operations, whose execution time is not known in advance. The function calls are actually handled as normal unbounded operations, allowing to use this approach with very complex call graphs. Unbounded operations are usually handled with an handshake mechanism, involving START and DONE signals. But the DONE signal is enough to infer the real ending time of the operation from the HCFT. Then it is possible to locate the precise time when the result of the unbounded operation has to be checked for the discrepancy.

3) Putting CFTs and OTs together: Having defined CFTs and OTs we can finally define SW and HW traces and their equivalence used to perform automated discrepancy analysis.

Definition 5. A *Software Trace* (ST) for a CDFG on a given input I is a pair $S(I) = (S_{CF}(I), O)$, where O is the list of Software OpTraces $S_O(BB_i)$ of the BBs in $S_{CF}(I)$. Similarly a *Hardware Trace* (HT) for an FSM on a given input I is a pair $H(I) = (H_{CF}(I), H)$ where H is the list of Hardware OpTraces $H_O(S_j)$ of the states in $H_{CF}(I)$.

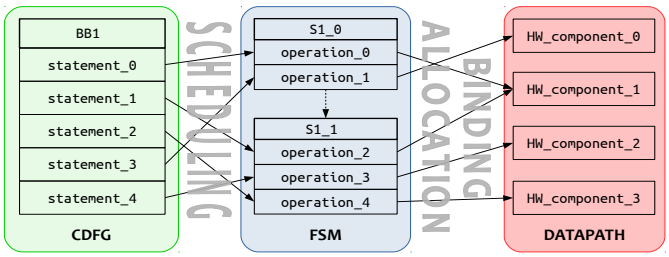

Fig. 2. Mapping between statements in a BB of the CDFG, operations in the corresponding states of the FSM and HW components in the DP.

Definition 6. Let $S(I) = (S_{CF}(I), O)$ be an ST for a CDFG and $H(I) = (H_{CF}(I), H)$ be the HT of the associated FSM. $S(I)$ and $H(i)$ are *equivalent* if

- $S_{CF}(I)$ is equivalent to $H_{CF}(I)$
- $\forall BB_i$ in $S_{CF}(I)$, being $\langle S_1, \ldots S_k\rangle$ its related states, $S_O(BB_i)$ is equivalent to $\langle H_O(S_1), \ldots H_O(S_k)\rangle$

IV. From Formal Description to Implementation

The description reported in section III-B is quite abstract and it was inspired by an early work in the field [15]. To show the effectiveness of the proposed methodology we created a debugger using our discrepancy analysis algorithm. The implementation has been developed extending the functionalities of the bambu HLS compiler [1]. Bambu takes C source files as inputs and generates Verilog [6] HW descriptions. The steps described here have been added to the default HLS flow of bambu. They are executed after the generation of the Verilog description of the HW to test, without affecting it in any way and without imposing any limitations on front–end optimizations on the options available for allocation/scheduling/binding in HLS. Instead, it makes full use of the HLS informations to select the signals and to perform the pattern matching between Software Traces and Hardware Traces.

A. Collecting STs and HTs

To adopt the approach described in section III-B, we need to retrieve the informations on the STs from the software and on the HTs from the generated HW. In particular, to generate the HTs it is necessary to detect the signals that needs to be selected in the generated HW.

1) Generation of the Software Traces: The SCFT is the list of BBs visited by the software execution, the SOT is the list of the results of assignment statements in the high–level source code. The data on the STs are collected by printing back the C source code after all the front–end optimizations performed by the HLS compiler. The assignments in the generated code are instrumented so that when the program is executed the assigned values are printed to a file, with additional high–level information. The C code–generator is designed to structure the generated source like the CDFG. The assignment are printed in Static Single Assignment (SSA) form, so every variable is assigned in only one operation. SSA ϕ operations [16] are splitted and substituted with assignments, with the approach described in [17]. In this way they can be

printed in the instrumented C code and they can be checked by the discrepancy analysis. There is no need to reorder the instruction in C to mirror the scheduling order of the operations in the FSM. This is due to the fact that every operation is executed only once in a BB and that it is scheduled in only one state among those corresponding to that BB. With this assumptions, having the operations printed in SSA form is enough to infer the correspondence between the assigned variable and the output signal of the HW component at the correct time. Moreover printing the C source code in SSA after the front–end optimizations allows to capture the internal variables inserted by the compiler. Executing the generated program is enough to retrieve both the SCFT and the SOT needed for discrepancy analysis.

2) Automated signal selection: The HCFT is represented by the value of the state signal in the controller of the FSM, while the HOT is essentially the set of values of the output signals of the HW modules to which the operations are bounded. For this reason, and for what said in paragraph III-B2, to use the approach described earlier the only signals to select are the following: 1) obviously the CLOCK signal; 2) the PRESENT_STATE signal of every FSM, which represents the HCFT; 3) the output signal of every HW module implementing a statement in the CDFG (this can also be a function call; the signal names are implementation specific); 4) START and DONE signals for HW module implementing unbounded operations. It is worth stressing that these signals are all we need to be able to perform the discrepancy analysis, even if the number of signals in the design is much larger.

3) Generation of the Hardware Traces: To generate HTs we rely on simulation. As said earlier, the simulation may not be the fastest way to obtain such information, but the slowdown is compensated by the automatic signal selection and discrepancy analysis. Moreover the simulation provides maximum observability of the signals, independently of the target technology. The design is executed with the same input as the C program and the signal variations are dumped in VCD format. To obtain the data needed to build the HCFT it is enough to collect the state signal of the controller of the FSM. The HOTs, instead, are composed using the values of the output signals of the HW modules used to implement the operations in the DP. The necessary signal are just a portion of all the signals in the design. The name of the interested signals can be inferred by the binding/allocation mapping built for the HLS process. The details of how this can be done are dependent on the specific HLS tool, but they are nothing more than the standard results of binding/allocation steps, without any customization. Once the interesting signals are identified, it is possible to restrict the VCD print only to them. With this signal selection the resulting VCD can be from about 35% to about 95% smaller than the original (see Fig. 6 in section V), with obvious benefits for the I/O time.

B. Matching STs and HTs

After the Hardware and Software Traces are collected the last step of our implementation performs automatic discrepancy analysis. The definitions already suggest how to compare the traces. CFTs can be easily compared with linear time complexity, simply iterating through the SCFT and verifying that the HCFT generated by the HW is equivalent, one element at a time. The linear complexity comes from the fact that the scheduling information used to compare BBs and states is built during the scheduling so it does not need additional calculations. Looking again at Fig. 1 we can deduce from the scheduling that BB0 is mapped onto the list of states (S0_0, S0_1, S0_2), BB1 is mapped onto (S1_0, S1_1), BB2 onto S2 and BB3 onto S3. If we start from the SCFT (BB0, BB0, BB1, BB2, BB3) and we substitute every BB with the corresponding list of states we obtain (S0_0, S0_1, S0_2, S0_0, S0_1, S0_2, S1_0, S1_1, S2, S3), which is indeed the HCFT. Then the two Control Flow Traces are equivalent.

The basic idea to compare OTs is very similar. In this case we take Fig. 2 as reference. The equivalence of OTs is done one BB at a time. Inside a single BB, the pattern matching proceeds along the list of FSM's states corresponding to the BB itself. For BB1 in the figure, the values assigned in the 5 statements must be checked. When the PRESENT_STATE signal of the FSM is in state S1_0 the value of the output signal of HW_component_0 must be equal to the value assigned in statement_3, and the value of the output of HW_component_1 must be equal to the value assigned in statement_0. Every variable is assigned in a single statement thanks to the SSA form, so this operation can be done. The same can be done for S1_1, to compare the remaining 3 operation.

This algorithm must be applied to all the BB in the SCFT. If the bug affects the control flow, the discrepancy is detected at the CFT level and the OTs comparison can be restricted to a single BB. This is done simply comparing the results of every statement with the value of the corresponding output signal of the component in the DP. Given that an operation is executed once for every BB, the time complexity to detect the bug in a single BB is $O(m)$, where m is the number of operations in the BB. The worst case scenario is when there is no CTF mismatch. In this case all the OTs for all the operations in all the basic blocks must be verified, because there could be a bug that does not change the control flow. This leads to a worst case time complexity of $\sum_{n=1}^{N} O(t(n))$ where N is the total number of operations and $t(n)$ is the number of times operation n is executed. Actually if a bug is present the comparison is faster, since it stops at the first mismatch.

V. Case Studies and Results

The approach described above have proved to be very versatile and efficient in automatic bug detection on different designs synthesized starting from C specifications using the bambu compiler. The discrepancy analysis is able to compare signals coming from pipelined and multi–cycle operations, with per–operations granularity even with very aggressive compiler optimizations. All the memory layouts available for HLS are supported. In this section we report in detail the results we collected during tests with our debugger.

A. Bug detection

The bugs detected from our debugger can be coarsely divided in three classes:

1) bugs already present in the original C specification;
2) bugs introduced using a library with flawed hardware components for HLS;
3) bugs introduced by the HLS tool.

Among bugs in (1), some are due to the quirks of the C language, like non–initialized variables or integer functions with `return` statements without value. These are allowed by the C standard, but they cause non–initialized signals to be set to "Z" in Verilog. Others are just real bugs caused by a wrong implementation of the specifications in the original C source. In this case the discrepancy analysis does not discover any mismatch between the high–level source code and the resulting HDL. However, given that this kind of bug are already present in the original code, all the well–known software debugging techniques can be used to find them, without involving the HW. Bugs in category (3) are actually bugs in the HLS tool. Like bugs in (2) they can be described in more detail based on how they affect the generated HW. In fact, their impact on the design is typically one of the following:

a) bugs in HW components used to implement operations;
b) bugs in the FSM controller logic;
c) bugs causing the design to loop or hang;
d) errors in the interconnection between components;
e) bit flips due to aggressive optimizations.

Bugs in all these categories have been successfully detected by our tool. Some of them were manually inserted for testing purposes, while others were actually found in the bambu HLS compiler. An out–of–bound bug was found in the `mips` benchmark in the CHStone HLS benchmark suite [18] (version 1.11_150204 and previous). Bugs of group (a) and (e) are the more frequent and they are detected with per–operation accuracy. When a mismatch occurs, the tool shows the position of the failing operation in the original C along with the mismatching signal and the timestamp. The failing operation may not be present in the original code, for instance if it was inserted by compiler optimizations. In this case, the debugger shows the information on the instrumented code after the optimizations. Bugs in the logic of the generated controller (b) can be due to wrong state optimizations of the FSM. They can cause a mismatch in the CFTs or bugs of type (c). In the latter case, the simulator can be set with a maximum number of cycles to simulate. Then the discrepancy analysis is performed on the partial traces. In this way the same method can be used to find bugs which normally would cause the design to hang or loop. Finally, bugs of categories (d) and (e) are actual compiler bugs. It is worth to remark that in our experiments we proved that with this approach it is possible to spot bugs which are not visible outside the design.

During our tests with the debugger we collected data about the coverage and the performance of our method, along with some other data on the advantages of the approach. Such results were collected during the execution of our discrepancy analysis on the circuits generated by the bambu compiler for the well–known set of CHStone HLS benchmarks, with different optimization options. In all the figures in the remainder of this section the names of the benchmarks are on the x axis, along with the optimization level.

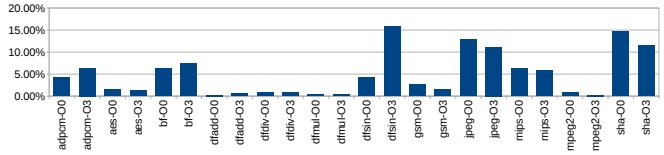

Fig. 3. Time overhead of discrepancy analysis, compared to simulation.

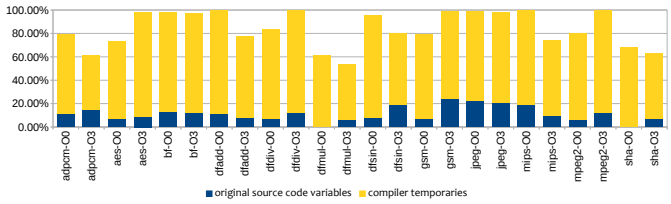

Fig. 4. Coverage: percentage of checked assignments at runtime.

B. Performance

In Fig. 3 we show the time overhead of our discrepancy analysis debugger, compared to the execution time of the simulation of the design being tested. The simulation was performed with the Modelsim SE–64 10.3 simulator. As we can see the discrepancy analysis represents a negligible overhead compared to the simulation: around 15% in the worst cases, but much less in most others. This is even more impressive if we think that normally the user would have to perform the analysis of the signals manually, without any knowledge of the circuit generated by HLS.

C. Coverage

Perhaps the most interesting results are about the coverage. In Fig. 4 we can see some coverage metrics about the granularity of the discrepancy analysis. The histogram shows the percentage of assignment at runtime checked with our method. The dark small area at the bottom represents the assignments involving variables which are present in the original source code. The larger yellow area on top, instead, represents the assignment involving compiler temporary variables. They are always at least 45% of the total, with a maximum of 89% for aes–O3. This clearly shows the potential of our method compared to other approaches to source–level debugging for HLS–generated circuits ([13], [12]), which are not able to check temporaries and force to disable compiler optimizations. It is also possible to see that the percentage of checked assignments is slightly affected by the optimizations, sometimes even showing an increase with more aggressive optimizations.

There are still some cases not covered by the methodology, namely checks on values resulting from pointer arithmetics. This is due to the different address spaces on the host machine of the SW and on the synthesized HW. As a result, for memory–intensive benchmarks the coverage is generally worse than others, because of the higher number of arithmetic

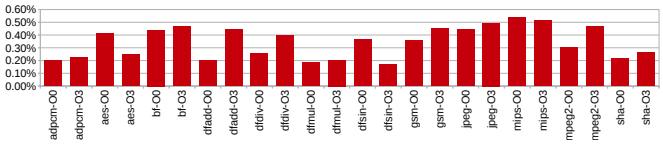

Fig. 5. Percentage of the selected signals on the total.

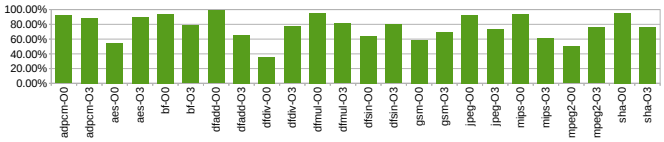

Fig. 6. Reduction of the VCD file size using signal selection.

operations between pointers. Work is ongoing to cover also the currently missed cases.

D. Other advantages

Another couple of interesting effects of our approach can be seen in the remaining figures. Fig. 5 shows the percentage of signals selected in the design with our approach. It is evidently very low. It also represents the number of signals needed to ensure HW/SW execution equivalence using our approach. Without automated signal selection, the user would typically need to find out the signals himself and to reconstruct the relationship with the original high–level source code. This would take a large amount of time especially if the user is not aware of the internal signal naming conventions of the HLS tool. This means that only the automatical signal selection is already a huge advantage for HW designers.

But there is more. With the signal selection we are sure that we have all the signals we need for the discrepancy analysis. Then there is no need to dump all the signals in the design to the VCD file. Only the interesting signals can be dumped, reducing the size of the generated VCDs. Fig. 6 shows the amount of the reduction of the generated VCD file size. In some cases this means bringing the size from some GBs to some MBs. This is also beneficial for the simulation time, which is always lower with signal selection, since the I/O bottleneck to print the VCD is less significant.

VI. Conclusions and Future Work

Significant progress has been made in recent years in the quality of circuits produced by HLS, but the support for debugging is still incomplete. The work presented here tries to bridge this gap proposing an approach for a completely transparent debugger which requires low effort for the user. This approach can help not only the software engineers with a limited knowledge of hardware design but also the hardware engineers to increase their own productivity.

The main contribution of the proposed technique is the automated discrepancy analysis, based on the selection of the signals in the generated HW, performed as well by the tool itself. The results on the coverage are very promising, showing that it is possible to find a very large percentage of the possible bugs also in complex scenarios with deep call graphs. Most importantly, such automated discrepancy analysis is able to find bugs related to temporary variables and no restrictions are imposed on compiler optimizations and memory layouts. Research is ongoing to complete the coverage also of addresses and pointer arithmetics.

References

[1] PandA project homepage. `http://panda.dei.polimi.it`.

[2] Y. Iskander, C. Patterson, and S. Craven, "Improved Abstractions and Turnaround Time for FPGA Design Validation and Debug," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, Sept 2011, pp. 518–523.

[3] E. Hung and S. J. Wilton, "Towards Simulator–like Observability for FPGAs: A Virtual Overlay Network for Trace–buffers," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13. New York, NY, USA: ACM, 2013, pp. 19–28.

[4] E. Hung, A. Jamal, and S. J. E. Wilton, "Maximum flow algorithms for maximum observability during FPGA debug," in *2013 International Conference on Field–Programmable Technology, FPT*, Kyoto, Japan, December 2013.

[5] E. Hung, T. Todman, and W. Luk, "Transparent insertion of latency–oblivious logic onto FPGAs," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014.

[6] "IEEE Standard Verilog Hardware Description Language," *IEEE Std 1364–2001*, 2001.

[7] J. S. Monson and B. Hutchings, "New approaches for in–system debug of behaviorally–synthesized FPGA circuits," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, September 2014.

[8] M. Ben Hammouda, P. Coussy, and L. Lagadec, "A design approach to automatically synthesize ANSI–C assertions during High–Level Synthesis of hardware accelerators," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, June 2014.

[9] J. Curreri, G. Stitt, and A. George, "High–level synthesis techniques for in–circuit assertion–based verification," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010.

[10] A. Ribon, B. Le Gal, C. Jego, and D. Dallet, "Assertion support in high–level synthesis design flow," in *Specification and Design Languages (FDL), 2011 Forum on*, September 2011.

[11] K. Hemmert, J. Tripp, B. Hutchings, and P. Jackson, "Source level debugger for the Sea Cucumber synthesizing compiler," in *Field–Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, April 2003, pp. 228–237.

[12] J. Goeders and S. Wilton, "Effective FPGA debug for high–level synthesis generated circuits," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, September 2014.

[13] N. Calagar, S. Brown, and J. Anderson, "Source–level debugging for FPGA high–level synthesis," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, September 2014.

[14] J. Cong and Z. Zhang, "An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation," in *Proceedings of the 43rd Annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: ACM, 2006, pp. 433–438.

[15] C.-T. Chen and K. Küçükçakar, "A source–level dynamic analysis methodology and tool for high–level synthesis," in *Proceedings of the 10th International Symposium on System Synthesis*, ser. ISSS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 134–140.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[17] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson, "Practical improvements to the construction and destruction of static single assignment form," *Softw. Pract. Exper.*, vol. 28, no. 8, pp. 859–881, July 1998.

[18] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "CHStone: A benchmark program suite for practical C–based high–level synthesis," in *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, May 2008, pp. 1192–1195.