POLITECNICO
MILANO 1863

Using Efficient Path Profiling to Optimize Memory Consumption of On-Chip Debugging for High-Level Synthesis

P. Fezzardi, M. Lattuada, F. Ferrandi

P. Fezzardi, M. Lattuada, and F. Ferrandi. Using Efficient Path Profiling to Optimize Memory Consumption of On-Chip Debugging for High-Level Synthesis. pages 1–19, 2017

The final publication is available via http://dx.doi.org/10.1145/3126564

Using Efficient Path Profiling to Optimize Memory Consumption of On-Chip Debugging for High-Level Synthesis

PIETRO FEZZARDI, MARCO LATTUADA, and FABRIZIO FERRANDI, Politecnico di Milano

High-Level Synthesis (HLS) for FPGAs is attracting popularity and is increasingly used to handle complex systems with multiple integrated components. To increase performance and efficiency, HLS flows now adopt several advanced optimization techniques. Aggressive optimizations and system level integration can cause the introduction of bugs that are only observable on-chip. Debugging support for circuits generated with HLS is receiving a considerable attention. Among the data that can be collected on chip for debugging, one of the most important is the state of the Finite State Machines (FSM) controlling the components of the circuit. However, this usually requires a large amount of memory to trace the behavior during the execution. This work proposes an approach that takes advantage of the HLS information and of the structure of the FSM to compress control flow traces and to integrate optimized components for on-chip debugging. The generated checkers analyze the FSM execution on-fly, automatically notifying when a bug is detected, localizing it and providing data about its cause. The traces are compressed using a software profiling technique, called Efficient Path Profiling (EPP), adapted for the debugging of hardware accelerators generated with HLS. With this technique, the size of the memory used to store control flow traces can be reduced up to 2 orders of magnitude, compared to state-of-the-art.

CCS Concepts: • **Hardware** → Methodologies for EDA; High-level and register-transfer level synthesis;

Additional Key Words and Phrases: High-Level Synthesis, On-Chip Debugging, Automated Bug Detection, Memory Optimization, Efficient Path Profiling

ACM Reference format:
Pietro Fezzardi, Marco Lattuada, and Fabrizio Ferrandi. 2017. Using Efficient Path Profiling to Optimize Memory Consumption of On-Chip Debugging for High-Level Synthesis. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (July 2017), 19 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

High-Level Synthesis (HLS) for Field Programmable Gate Arrays (FPGAs) aims at the automatic generation of hardware designs from algorithmic descriptions, and is gaining popularity in various fields of computing and electronic design. HLS is seen as a way to reach new fields of application and to enlarge FPGA market, increasing designers' productivity and reducing time-to-market. It is also a key technology to abstract away the details of the Register Transfer Level (RTL) description, bringing FPGAs to software engineers.

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue.
Authors' emails: {pietro.fezzardi, marco.lattuada, fabrizio.ferrandi}@polimi.it.
 1539-9087/2017/7-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

The improvements and the optimizations introduced in the High-Level Synthesis methodologies have increased the distance between the original high-level specification and the final RTL, reducing their correlations. If a hardware accelerator exhibits misbehaviors, either in simulation or on-chip, finding the cause can be cumbersome and time-consuming. This is even more critical for HLS-generated circuits, where the generated hardware description is not intended to be human-readable. Since more than 50% of the time spent on average in a project is for verification [10], debugging support for HLS tools and for the generated designs is attracting great interest.

Debugging techniques for HLS can be divided in two groups, with intrinsic advantages and weaknesses: I) techniques implemented on-chip; II) techniques based on simulation. On one hand, simulation seamlessly provides full observability of the circuit during the entire execution, and it does not require architectural modifications. On the other hand, in-circuit debugging is the only way to capture interactions with other components of the system. In real world applications, designs are usually composed of many components, which can be generated by means of HLS, provided by third parties as Intellectual Property (IP) blocks, or hand-coded by the designer. IPs provided by vendors may not have been tested for some corner cases of the end users, and hand-written Hardware Description Language (HDL) descriptions may yield different results in simulation and after synthesis [18] [20]. This scenario further complicates debugging when HLS is used for system level design with integration of third parties IPs, and calls for a system level methodology to debug HLS-generated systems directly on-chip. Many recent results have been pushing the limits of debug capabilities for FPGAs and designs generated with HLS. To this end, one of the most important data which must be collected during the on-chip execution is the state signal of all the Finite State Machines (FSMs) of the circuits. Most of the existing approaches includes this in the extracted traces [5], to infer some of the information to be provided to the users [8], or to decide how to trace and analyze the bugs of the whole circuit [14].

This work presents a methodology for the integration of components for online automated debugging of HLS-generated circuits. The main contributions are:

- The customization and optimization of the Efficient Path Profiling [2] to compress information about FSM execution traces, minimizing their memory footprint on FPGA.
- The proposal of a HLS flow to generate control flow debugging components, exploiting the enhanced Efficient Path Profiling, for automatic runtime bug detection on-chip.

The proposed flow generates a golden reference from the high-level source code along with the HLS information and generates and inserts control flow checkers in the design. The on-chip control flow checkers analyze the behavior of the design at runtime, notifying during execution if a bug occurs and providing information on its sources. These characteristics are completed by automated back-tracking of the detected bug to the original source code, using the data sent off-chip by the checkers when a bug is detected.

In the following, Section 2 presents the related work, while Section 3 sets the background for the work: Discrepancy Analysis HLS debug flow and Efficient Path Profiling [2]. Section 4 describes the proposed methodology and debug flow, the modification of Efficient Path Profiling to target HLS debugging, and how to optimize it to reduce memory usage. The results are discusses in Section 5. Finally Section 6 closes and outlines possible future work.

2 RELATED WORK

There are several different ongoing efforts to endow HLS frameworks with effective support for debugging, both in academia and in industry [15] [24] [29]. In general, the proposed approaches can be roughly divided into two main classes:

(1) Approaches which use HLS information for automatic and efficient implementation of debugging components, embedded in the designs to provide observability and to trace the behavior of the circuits during execution.
(2) Approaches based on the instrumentation and execution of the Intermediate Representation (IR) of the HLS compiler or of the original source code, which is used to generate a golden reference for the behavior of the circuit. This golden reference is then used to check automatically if the design is working properly at a functional level, and to report useful source-level information to the users.

An example of the first type of approach has been proposed by Monson and Hutchings [21][23], which use source level transformations to insert tracing logic (Event Observability Port and Buffers) for the output signals of operations. The authors ported the methodology on-chip in [22], but only focusing on Datapath operations, without detecting bugs involving control flow. Other approaches instead are based on the tracing of the state signals of the FSMs performed directly on-chip. Goeders and Wilton [12][13][14] generate a component aimed at managing debugging and at saving the execution traces on FPGA. They show different techniques to reduce the memory usage necessary to store the traces at runtime, and to support compiler optimizations. Their focus is on providing a software-like debug framework, where users can manually inspect the traces after execution or can suspend the hardware (HW) to analyze its state. They do not provide automated bug detection, unlike the methodology proposed in this work. They also note that suspending the execution may break interactions with other components of the system and potentially introduce other bugs.

Campbell et al. [5], instead, focus on Application Specific Integrated Circuits (ASICs) and adopt an hybrid approach. They generate both a golden reference for the hardware execution from HLS IR, and a set of components that are used to extract the equivalent execution traces from the circuit (called hardware signature). The golden reference and the hardware signature are compared at the end of the execution and bugs are automatically detected. Iskander et al. [19] propose a different hybrid approach composed by two parts: a High-Level Validation and Low Level Debug. For the High-Level Validation they run the golden reference software on a softcore on the FPGA, saving the results and comparing them with the results obtained from the accelerators. The main intent of this stage is to create a workflow that is easily embeddable in automated regression testing and unit testing. The authors say that the memory and logic footprint of the High-Level Validation is high, but they do not report data since they consider it acceptable for the scenario of unit testing. The Low Level Debug, instead, uses partial reconfigurability to provide observability, insert breakpoints and provide a software-like debugging experience.

Another trend in on-chip debugging is based on bringing ANSI-C assertions to HW [6] [16] [26], adding assertion checker circuits. The FSM of the checker can be executed concurrently to the controlled module [6] [16] or the synchronization can be directly performed by the FSM of the accelerator itself [26]. 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 an assertion spots a wrong condition, the real root cause can be a previous bug which is difficult to find. Finally, if the circuit happens to enter in a hanging state, the relevant assertion trigger point may be never reached at all.

HLS of assertion checkers is also used for Runtime Verification. Selyunin et. al [27] use HLS to generate runtime verification checkers in automotive chip design. Runtime Verification differs from the automated bug detection proposed in this paper because it generates checkers for temporal logic properties that must hold for all the possible executions of the circuit. Another difference is that in runtime verification the properties to be checked must be specified in some way by the

designer. This work, instead, automatically generates the checkers without requiring the designer to specify the properties they have to check. The generated checkers are only capable of guaranteeing equivalence between hardware and software execution on a fixed input. While this may seem a limitation, the final goals of the two approaches are very different. Runtime verification checkers guarantee that some properties hold during all the lifetime of the checked system, and they are even embedded in final products. The proposed approach, instead, helps HLS developers to efficiently and accurately find bugs, without the need of specifying temporal logic properties and automatically backtracks bugs to the original source code.

Several other works adopt methodologies based on software instrumentation or simulation. Campbell et al. [4] adapt their methodology to FPGA, but differently from [5] they rely on simulation to generate the hardware signatures. For this reason the design runs unmodified and there are no issues with on-chip memory usage. The same approach is adopted by Fezzardi et al. [8][9], who define two complementary levels of the checks performed by their methodology: Control Flow Level and Operation Level. They also say that their methodology should be applicable on-chip, but in their work they still use simulation. Yang et al. [30][31] use the golden reference obtained from the IR to insert instrumentation in the the RTL, but the whole debugging flow still relies on simulation. However, unlike [4] and [8], the comparison between hardware and software is not performed at the end of the execution, but directly by the RTL instrumentations during simulation.

The main limit of all these methods based on simulation is the impossibility of detecting post-synthesis bugs. On the contrary, Calagar et al. [3] analyze automatically the discrepancies between hardware and software, but they do it online, during the on-chip application execution. Their work exploits both simulation and on-chip debugging. They do not generate the golden reference in advance, but instead they use *gdb* server to analyze the software, the simulator APIs to analyze the simulated RTL, and Altera SignalTap for in-circuit debugging. However, they do not support most of the compiler optimizations performed during the HLS and the use of SignalTap causes a high memory usage for the buffers used to collect the traces, as reported also in [21].

3 BACKGROUND

This section provides the background for the methodology proposed in this paper. Specifically, Section 3.1 describes the Discrepancy Analysis debug flow used for offline debugging of HLS-generated accelerators, and Section 3.2 presents the Efficient Path Profiling.

3.1 Discrepancy Analysis Debug Flow

As described in Section 2, one of the key ideas for effective debug of HLS-generated designs consists of extracting from the high-level specification a golden reference for the behavior of the circuit, that is then used to automatically find bugs in the generated accelerators. This method, used for example in [4] [9] [30], is often referred as Discrepancy Analysis [3] [8]. Figure 1 depicts an example of workflow for Discrepancy Analysis (DA) similar to what is described in [4] and in [8]. The original high-level specification, typically written in C, is the input of the HLS. The HLS then produces two outputs: the HDL of the circuit (on the left), and some form of executable instrumented IR (on the right). This

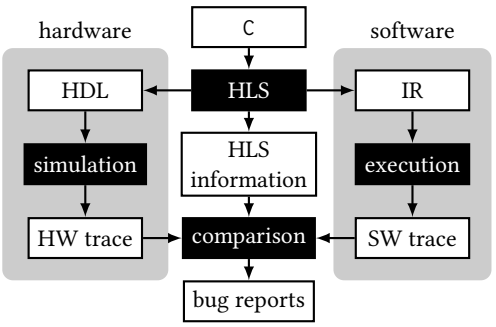

Fig. 1. Outline of the Discrepancy Analysis debug flow.

is usually possible thanks to Just-In-Time compilers that are able to execute the IR, or by printing back the source code with instrumentation after HLS optimizations. The *simulation* of HDL then generates the HW traces while the *execution* of the IR generates the SW traces, which represent the golden reference for correct execution. Finally the *comparison* of traces is performed completely automatically, detecting the bugs that are reported associating them with the corresponding high-level source code.

Several small variations of this flow can be found in literature. In [3] the *comparison* is executed on-fly during hardware and software executions. The main other flavor of DA (used for example in [31]) uses the SW traces with HLS information to create debugging components to be embedded in the design. These components are executed in simulation along with the rest of the design to automatically find bugs.

3.2 Efficient Path Profiling

Tracing the control flow of designs generated with HLS corresponds to observing the state signals of the FSMs, since they are typically built starting from Control Flow Graphs (CFGs) [1], which in turn represent the structure of the high-level specifications. Every node of a CFG is called Basic Block (BB) and it contains a list of instructions that are executed sequentially in software. The edges describe the branches and the loops. In this way, the CFG statically represents all the possible paths of execution of the software at runtime. The dynamic information about executed paths can be collected by means of a software profiling technique called Efficient Path Profiling (EPP) [2]. EPP is typically used to collect runtime information about paths in a Control Flow Graph, but in Section 4 it will be adapted to hardware.

```
BB1 | cond = a > 0;
    | if(in1)
BB2 |    target = a;
    | else
BB3 |    target = init();
BB4 | while(target != current && iter < 10){
BB5 |    iter++;
    |    if(current < target)
BB6 |       current = pow(current,2);
    |    else;
BB7 |       current *= coeff;
BB8 |    temp[iter] = current;
    | }
BB9 | return current;
```

Fig. 2. Example of source code to be synthesized. Basic Blocks identifiers are displayed on the left.

Intuitively, a path is a sequence of Basic Blocks executed consecutively. Since one specific execution of a function is a sequence of Basic Blocks, it can be efficiently described by means of a path. However, if the Control Flow Graph of a function contains at least one uncountable loop, the number of possible paths which can be executed is potentially infinite, since the loop can be repeated an arbitrary number of times. To overcome this issue, the set of paths which can be extracted from a Control Flow Graph must be restricted, so that the execution trace of a function is described by means of a sequence of paths. Ball and Laurus, in their seminal paper on EPP [2], proposed a possible restriction to the paths which can be extracted from a Control Flow Graph and an efficient technique to compute and compress information about them.

Figure 2 shows the source of the example used in the rest of this Section to describe the Efficient Path Profiling. The corresponding CFG is shown in Figure 3. The function contains a loop [28] composed of the Basic Blocks $\langle BB4, BB5, BB6, BB7, BB8\rangle$. The only feedback edge (i.e., the edge which closes a cyclic path with origin in the *BBEntry*, see [28]) is $\langle BB8, BB4\rangle$.

The paths considered valid by Ball and Laurus are all the acyclic paths $BB_i,\ldots,BB_j$ such that:

(1) BB_i is the *BBEntry* of the Control Flow Graph or the target of a feedback edge;
(2) BB_j is the *BBExit* of the Control Flow Graph or the source of a feedback edge.

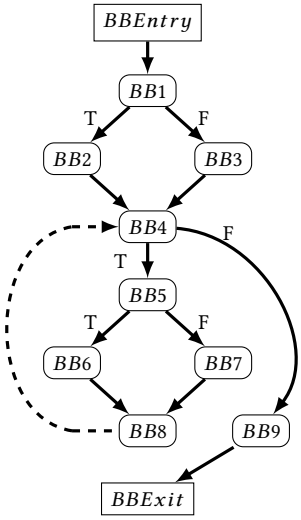

Fig. 3. The Control Flow Graph of the example in Figure 2. The dashed edge is the only feedback edge.

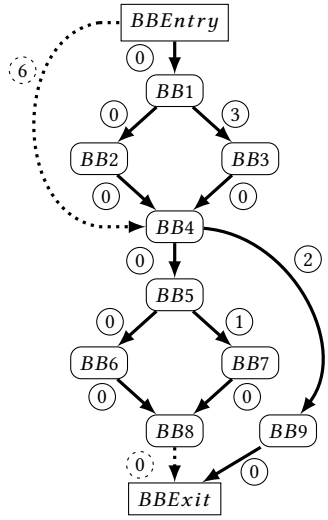

Fig. 4. The Path Graph of the example in Figure 2. The dotted edges are the additional auxiliary edges.

Id	Path
0	*BBEntry BB1 BB2 BB4 BB5 BB6 BB8 (BBExit)*
1	*BBEntry BB1 BB2 BB4 BB5 BB7 BB8 (BBExit)*
2	*BBEntry BB1 BB2 BB4 BB9 BBExit*
3	*BBEntry BB1 BB3 BB4 BB5 BB6 BB8 (BBExit)*
4	*BBEntry BB1 BB3 BB4 BB5 BB7 BB8 (BBExit)*
5	*BBEntry BB1 BB3 BB4 BB9 BBExit*
6	*(BBEntry) BB4 BB5 BB6 BB8*
7	*(BBEntry) BB4 BB5 BB7 BB8*
8	*(BBEntry) BB4 BB9 BBExit*

Table 1. Valid paths of the Control Flow Graph in Figure 3.

This is modeled by building a modified version of the Control Flow Graph: the Path Graph. The changes to apply to the CFG are: add an auxiliary edge from *BBEntry* to the target of each feedback edge; add an auxiliary edge from the source of each feedback edge to *BBExit*; remove all the feedback edges. The valid paths of the CFG correspond to the paths from *BBEntry* to *BBExit* in the Path Graph. The Path Graph derived from the Control Flow Graph in Figure 3 is shown in Figure 4. Dotted edges are the added auxiliary edges.

Let N be the number of paths in the Path Graph, Efficient Path Profiling uses the number from 0 to $N-1$ to identify them. It also associates a weight $W_{i,j}$ (also called edge increment) to each edge $\langle BB_i, BB_j \rangle$, so that the identifier of a path is equal to the sum of the weights of the edges which compose it. In Figure 4 the edges are labeled with the weights computed with EPP. According to these weights, each path is associated with an identifier from 0 to 8. Table 1 lists the identifiers for the valid paths in Figure 4. As an example, the execution trace $\langle BBEntry, BB1, BB2, BB4, BB5, BB6, BB8, BB4, BB9, BBExit \rangle$ can be compressed in the sequence of paths $\langle 0, 8 \rangle$ without loss of information. For the details of how path identifiers and edge weights are computed see [2].

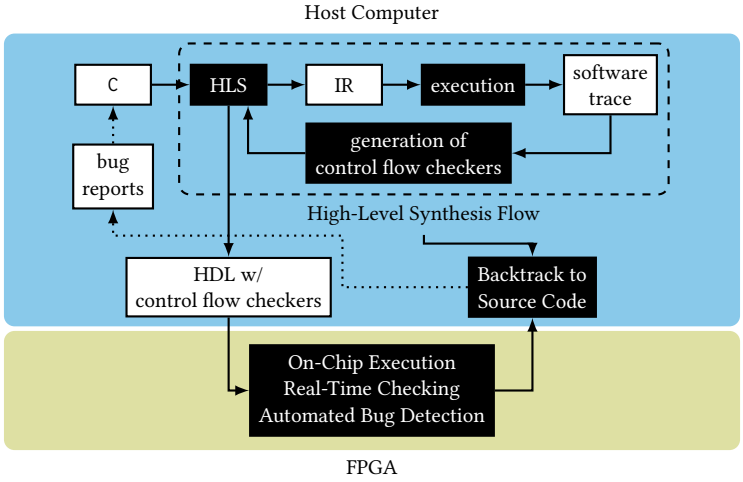

Fig. 5. Modified Discrepancy Analysis flow for on-chip debugging.

In adopting this technique, there are three advantages that are relevant for this work:

(1) the number of bits necessary to represent paths is minimal;
(2) at every point in the execution, the information about the currently executed path is represented by a single integer (i.e., it can be put in a register);
(3) to update the counter used to store the currently executed path it is only required to increment a local variable by the weight of the last traversed edge.

4 USING EPP WITH ON-CHIP DISCREPANCY ANALYSIS TO IMPROVE HLS DEBUGGING

This work aims at adapting a Discrepancy Analysis flow to debug HLS-generated hardware on FPGA, while reducing the memory usage. In particular, the focus is on control flow, i.e. on the state of the FSMs that control the generated hardware. In this respect, the proposed approach is applicable to all the HLS flows that generate functional modules composed of a Finite State Machine and a Datapath. It may not be applicable to other models, like HLS of streaming computations. This section describes in detail the different aspects of the methodology. Section 4.1 explains how the Discrepancy Analysis debug flow is modified to generate the debugging components, and to integrate them in the design. Section 4.2 describes how the reference traces for the generated design are computed starting from software execution. Section 4.3 shows how Efficient Path Profiling is adapted to the debugging of FSMs generated with HLS. Section 4.4 illustrates how to guarantee that the first bug is correctly identified on-chip, while Section 4.5 sketches the functionalities of the control flow checkers. Section 4.6 discusses how to compress the execution traces to reduce the memory usage.

4.1 Modified Discrepancy Analysis Flow for On-Chip Debugging for HLS

For this work, the Discrepancy Analysis debug flow shown in Figure 1 has been modified to run on-chip. The new flow is depicted in Figure 5. The portion on top with the blue background runs on the host computer, while the portion below with the green background executes directly on the FPGA. The High-Level Synthesis flow, enclosed in the dashed box, has been extended to use the

software traces to generate dedicated components, called *control flow checkers*, that perform the control flow checks on-chip.

A customized instance of control flow checker is integrated alongside the FSM of every functional module. This enables a fine-grained customization of the checkers, to use the smallest number of bits necessary for every function. The optimal number of bits and the dimension of the necessary memory depend also on the software reference trace. Since all these factors can be evaluated on the IR before the generation of the checkers, it is possible to explore the optimal values of the parameters for every checker before synthesis, as explained in Section 4.6. Control flow checkers contain memories that are initialized with the expected control flow trace, computed from software with the modified version of Efficient Path Profiling (see Section 4.2). This is one of the differences between previous works and the methodology proposed here: the debugging logic is not used to memorize information on the HW execution, but to compare it in real-time with a golden reference. Others have adopted this strategy [4][31] relying on simulation instead of debugging on-chip. In this work, the debugging components are integrated in the design by the HLS engine and then synthesized and executed on FPGA. As soon as a checker detects a mismatch between the expected execution and the real behavior, it notifies it to the host. Only a limited amount of information is exchanged: a unique identifier of the checker instance (that is uniquely determined during HLS and embedded in every checker), and the offset in the trace execution where the mismatch happens.

With this approach, the quality of debugging is at least as good as with simulation-based checkers. This means that all the control flow bugs detected with simulation are also visible with the hardware checkers, while with on-chip debugging it is potentially possible to detect post-synthesis bugs and mismatches that come from problems introduced by system integration. The proposed methodology allows to reduce the memory necessary for the traces while providing two advantages: (1) improving visibility of the bugs that only arise on-chip; (2) helping designers to automatically backtrack bugs to the original source code. The first point is an advantage compared to approaches that only rely on simulation [4] [31]. The second is not possible with other approaches that focus only on providing architectural support for the collection of the traces [12], leaving the burden of their manual analysis entirely to users.

4.2 EPP for Hardware Trace Generation

The main idea for the design of the checkers is to keep the reference traces small, in order to minimize the memory usage. To this end, Efficient Path Profiling must be adapted to work on FSMs. The advantage of EPP is that a single path represents a list of Basic Blocks. In most cases, storing a path identifier is cheaper than storing the list of identifiers of BBs that compose that same path. To use EPP on Finite State Machines it is necessary to rely on information extracted from the HLS process. During HLS, the Control Flow Graph of the original source code is translated into a FSM. The precise scheduling of the single operations is not really relevant here, because the focus is on control flow. What is important is that during the HLS every BB is mapped onto a consecutive list of states in the FSM, like depicted in Figure 6. This mapping is called M in the following and it has some useful properties.

- For every Basic Block BB_i in the CFG, there is one and only one ordered sequence of connected states in the FSM such that $M(BB_i) = \langle S_{i,1}, \ldots, S_{i,n} \rangle$.
- As a consequence, for every given path on the Control Flow Graph $p = \langle BB_i, \ldots, BB_j \rangle$, it exists one and only one path on the FSM $p' = M(p) = \langle M(BB_i), \ldots, M(BB_j) \rangle$.

This intuitively means that the CFG and the FSM have the same branch structures. Thanks to these properties and to the algorithm used in EPP for path numbering and edge weight computation, EPP can be used without modifications also on the Finite State Machine. This guarantees that the

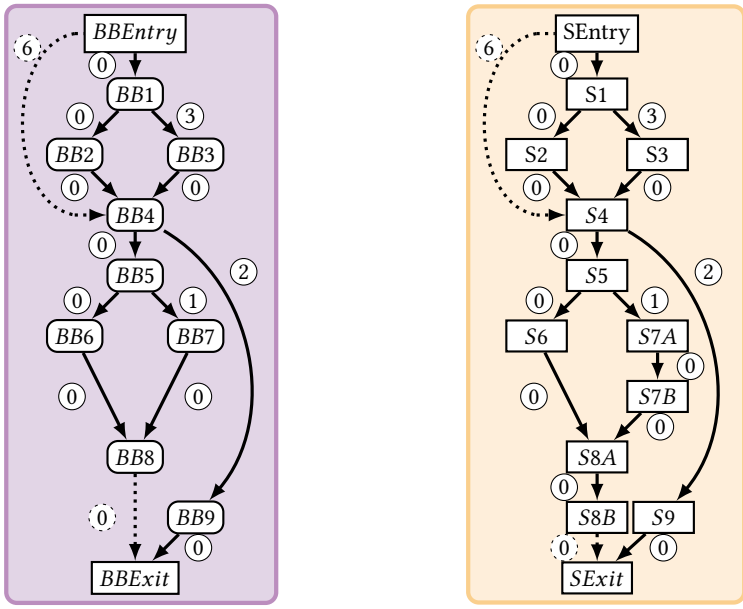

Fig. 6. Two Path Graphs obtained applying Efficient Path Profiling respectively to the Control Flow Graph of Figure 3, and to its associated Finite State Machine. Notice the similarities between the graphs, and the equivalence of the edge increments.

algorithms on the CFG and on the FSM calculate the same identifiers for every path p and for the associated path $p' = M(p)$. Finally, in the CFG the edges with weight $W \neq 0$ are only the outgoing edges from BB with branches. This means that for every Basic Block BB_i the edges on the FSM that are internal to $M(BB_i)$ will always have weight 0.

All these properties make possible to compute the expected list of paths identifiers for the hardware simply starting from software execution. EPP is used on the CFG of the software, instrumented to print the list of identifiers of the traversed paths, that can be directly used as a golden reference trace for the generated hardware.

4.3 EPP for Finite State Machines

In Sections 3.2 and 4.2 the discussion is focused on Path Graphs, computation of the edge weights for EPP on FSMs, and how to generate the golden reference from software. However, during the actual execution, the hardware follows the Finite State Machine, not the Path Graph. Feedback edges, that were excluded by the computation of the increments, can be taken during execution. In [2], Ball and Laurus select a minimal set of edges in the Control Flow Graph and add instrumentations to increment and reset the EPP counter along the edges, including feedback edges. In FSMs, the edges model the state transitions. For effective debugging, the structure of the FSM generated by HLS cannot be altered adding new states or transitions. For this reason increments, resets and checks must be scheduled in the existing states. In the following, the discussion is simplified describing the instrumentations as if they were actually inserted in the FSM. However, bear in mind that they are actually isolated from the FSM and completely encapsulated in the checker component. The FSM only exposes its current and next state to the checker as explained in Section 4.5.

To understand where to insert instrumentations in the FSM, it is useful to make a comparison with software. In software, ensuring that the current execution matches the expected path means ensuring that the path up to that point is correct. To do this, it is enough to check that the counter used to accumulate edge weights matches the identifier of the expected path. Ideally, to have a strong guarantee it would be necessary to check this condition at every cycle, but in practice this is not necessary. Indeed, the only places where the execution path may diverge from the expected are branches and feedback edges. Given that path identifiers are unique, it is enough to check that the current path is correct only on feedback edges and upon the termination of the execution of a function. In hardware, this means that the checks must be performed in states that are destination of feedback edges, and in final states of the FSM (i.e., states representing the termination of the execution of the associated function). In software the path counter can be checked after the termination of the function and before returning control to the caller. In hardware, instead, the check is anticipated to the final state itself. This is possible thanks to the fact that a final state of a FSM has no outgoing edge, which means that the last increment on the EPP counter is computed in the previous cycle and its final value is already available.

Feedback edges have to be treated separately. The reason is that the EPP counter is reset on feedback edges to the value of the weight of the auxiliary edge connecting the entry state to the destination state of the feedback edge. At the same time, the checker must ensure that the path before taking the edge was correct, in the first state after the feedback edge. In order to do so, the path identifier before the feedback edge must be registered, and the check must be deferred to the following clock cycle.

4.4 Detection of the First Mismatch

One of the goals of DA is to automatically detect the first mismatch between hardware and software execution. With simulation this property is easy to achieve: the simulated design executes concurrently, but the debugger can analyze the whole traces and determine the first mismatch. To keep this property on hardware it is necessary that checkers can spot a mismatch with a latency of a single cycle. This is one of the subtle differences between EPP for software and for hardware. The main reason is that software runs sequentially, and only a single function is in execution at any given time. Thus, if a mismatch is detected it is clearly the first. This is not necessarily true on concurrent hardware when multiple FSMs execute concurrently, which happens quite often in HLS-generated accelerators. The reason is that function calls are modeled with concurrent communicating FSMs. An handshaking mechanism between caller and callee allows the caller to wait in an idle state until completion of the callee.

Consider for example the FSM in Figure 6, and suppose that a call to another function is scheduled in state S2. The additional idle state is not depicted here to avoid to overcrowd the picture. The function call is executed conditionally, only if a certain branch is taken. Assume now that the expected path identifier calculated with EPP is ⟨5⟩, meaning that the expected states executed by the FSM are ⟨S1, S3, S4, S9⟩. In this situation, according to what explained in Section 4.3, the control flow checker would wait to check the execution path until state S9, because no feedback edge is taken. However, if the control flow diverges earlier and it takes the wrong branch from S1, this would result in the execution of S2 instead of S3, triggering the function call scheduled in S2. Given that the called function was not expected to execute, its associated control flow checker would detect a failure, but the root cause is actually a failure in the caller. In this scenario, if the checks are limited to what described in Section 4.3, the detection of the first bug and the automatic identification of the cause would be wrong.

This problem can be circumvented by ensuring that the running execution path is always checked in all the states with function calls. Doing this makes possible to catch control flow mismatches in the current scope before passing control to other FSMs. The mismatch is hence detected in the proper location in all the cases. If the mismatch is detected in the caller, this means that the path has diverged before the call and the current EPP counter can be used to identify the origin of the divergence. If the mismatch is detected by the callee, instead, it is guaranteed that the caller control flow was correct until the call, ensuring that any mismatch detected by the checker in the callee is actually located in that FSM.

4.5 Architecture of the Control Flow Checkers

A dedicated instance of control flow checker is created for the FSM of every function. All the functionalities described in Section 4.3, Section 4.4, and Section 4.6 are implemented as a single component. The checker is separated from the FSM, which is not altered by the instrumentation. The only signals used by the checker are the input and output signal of the state register of the FSM. They are called respectively `next_state` and `present_state` in the following and they drive all the operations of the checker. However, directly using the `next_state` signal may have timing implications, because it is likely to place the checker on a potential critical path. To avoid problems the inputs of the checker are registered, so that all the operations of the checker are executed with a delay of one cycle. This does not prevent the methodology to correctly identify the first fault, because all the checkers are subject to this delay, guaranteeing that the first mismatch notified outside is correct.

Every checker contains a read-only memory, called *trace memory*, initialized with the expected EPP trace. It is a single port memory, accessed with constant fixed alignment and a registered address: `cur_off`. The value in the *trace memory* at `cur_off` is next entry in the EPP trace, containing the identifier of the next expected path. This identifier is also stored in a register called *prev_trace*, used to check feedback edges that are delayed by one cycle as described in Section 4.3. A second read-only memory, called *increments memory*, contains the edge increments and is addressed using the union of `present_state` and `next_state`. The value of the edge increment read from this memory is added at every cycle to a register holding the current EPP counter accumulator. This EPP counter is compared directly with the value in the *trace memory* at offset `cur_off`, for states that end a path. The value of EPP counter is also reset to 0 on feedback edges, after recording it for the delayed check with the mechanism described in Section 4.3. The stored value is compared in the next cycle with the value of *prev_trace*. This completes the checking mechanism.

It is worth to notice that theoretically the dimension of the *increments memory* would be quadratic with the number of the states of the FSM. However, this memory contains very sparse data, since present_state and next_state can only represent valid transitions of the FSM. In addition, even on valid edges, most of the increments computed by EPP are zero, because only states before branch instructions have outgoing edges with weights $\neq 0$. Hence, the number of increments stored in this memory is actually $\sum_{s \in FSM}(out\_degree(s) - 1)$, where the *out_degree* of a state is the number of outgoing edges. The term of this summation are actually zero for every state s with only one outgoing edge. Practically, this means that the synthesis tool will optimize it and will implement it using combinational logic instead of Block RAM (BRAM).

What remains is the notification mechanism, by means of which the checker notifies to the outer world when it finds a mismatch. The checker detects if the mismatch is related to the current state or to a delayed check on a feedback edge. If both kinds of mismatches are detected in the same cycle, the mismatch related to the delayed check is notified, because it actually happened in the previous cycle. After the selection, the checker writes on output signals the following three data: a bit

asserting that a fault was detected, an identifier determined at design time that uniquely identifies the hardware scope where it was detected, and the offset in the EPP trace where that happened. Using this information, the debugger running on the host machine can unroll the execution traces, map them on the CFG and the FSM, and backtrack the fault to the original source code using HLS information.

It is possible to estimate the area required for a checker in terms of BRAMs and logic. Let $trace\_nbits$ be $log_2(PathMax)$, where $PathMax$ is the largest path identifier computed by EPP for the checked FSM, and off_nbits be $log_2(trace\_len)$, where $trace\_len$ is the length of the golden reference trace for the checker.

The main contribution to area on FPGA is given by the BRAMs used for the traces. The number of required memory bits for the trace is $checker\_mem\_bits = trace\_len \times trace\_nbits$. This value can be used to compute the total number $NRAM$ of BRAMs that will be used on FPGA, but the result depends on the size of the memories available on the specific target platform, and on the total number of checkers that are added to the design. To give a worst case estimation it has to be assumed that every trace needs its own BRAM, hence $NRAM = \sum_{checkers}(ceil(checker\_n\_bits/BRAM\_size))$. If the traces are small and the available BRAMs are true dual port memories, the actual number could be lower because two traces can be stored on a single memory block. This strictly depends on the device and on how the memories are inferred by the synthesis tools.

For what concerns the combinational logic necessary for the checker, the actual area occupied on FPGA depends on the target device and on the results of technology mapping and place-and-route. However, this is the list of elementary components necessary for the checker: 3 registers with width of $trace\_nbits$; 1 adder with width of $trace\_nbits$; 1 register and 1 incrementer with width of off_nbits; 2 comparators with width of $trace\_nbits$. The state machine of the checker is very simple: the state register is only one bit. In addition, there are some bitwise logic gates and a off_nbits wide multiplexer for the notifier. What cannot be estimated is the area of the *increments memory* because it is tightly dependent on the structure of the checked FSM and on the sparsity of transitions. The same holds for frequency estimation, because the critical paths use the registered inputs of the checker and the *increments memory* to compute the new `curr_off` that is then used to address the *trace memory*. Anyways, the critical path is mainly dominated by the memory latency, so the maximum clock frequency for a checker is not far from the BRAM maximum frequency.

4.6 Optimization of Memory Usage

The strategy of adapting EPP for debugging of FSMs explained in Section 4.2 already shows advantages compared to the state-of-the-art (see Section 5 for a detailed discussion). This is a consequence of the fact that previous approaches [12] typically encoded control flow traces as list of states, and then focused on their compression. EPP, instead, represents an entire path with a single identifier. Hence, a single entry in an EPP-encoded trace already represents a compressed list of states without loss of information. This is enough to give better baseline memory usage in most cases (see discussion in Section 5), but there are some situations where this is not necessarily true. This happens when the FSM contains a large number of branches and loops, in proportion to the number of states. In such situations two situations can occur: 1) the total number of valid paths computed by EPP is high, possibly higher than the number of states; 2) the number of times each loop is executed is also high. When both these circumstances are true, the result is that every single entry in the EPP trace is large (because of the large number of possible paths) and every entry is repeated multiple times in the trace (because of the large number of loop iterations).

In these cases, storing the traces as lists of states can be more efficient than EPP. The reason is that if the number of states is smaller than the number of paths they can be represented with

fewer bits. This is especially useful if the bodies of the loops are composed of short lists of states, because a short list of states may be stored in fewer bits than a single path. This advantage is less significant if the bodies of the loops are long list of states. However, Goeders et al. [12] show a simple strategy that can be used to compress traces of states specifically in this case. They notice that in the inner loop body the FSM typically traverses a list of states with consecutive identifiers. So they add a fixed number m of metadata bits to every trace entry. These bits are used to store the number of consecutive cycles for which the FSM state simply increases by one. This allows to compress a serial list of states, representing each loop iteration in a single packed entry of the state trace with metadata. This strategy cannot be used with EPP, because it already encodes an iteration of a loop in a single entry in the trace. Instead, with EPP the metadata can be used more effectively to compress multiple consecutive iterations, as explained later in this Section. In [12] the actual size of the metadata is determined with profiling and is fixed for all the FSMs. This approach has a limitation: it directly depends on the encoding of the state signal and the ordering of the states. If the state signal is one-hot encoded, every entry in the state trace is very large and it may need to be re-encoded in a smaller format to reduce the traces. Moreover, to support the compression scheme described in [12] it may be necessary to change the state enumeration. These operations are not necessary with EPP encoding.

EPP does not suffer of this drawback, because the edge increments are not dependent on the FSM encoding or ordering. However, the problem of compression of multiple iterations of loops with high path weights is still relevant. A loop iteration is represented by a single entry in the EPP trace. Hence, multiple iterations of the same loop are lists of repeated entries, each one representing an iteration. A simple way to compress this kind of traces is to append a fixed number of bits containing metadata to every trace entry. For every entry in the EPP trace, the value of the metadata represents the number of times the path is repeated after the first time. In this way, every entry in the EPP trace can represent up to 2^k iterations of the same path, where k is the number of bits used for the metadata. Moreover, the optimal value for k can be determined before the generation of the control flow checker. This is a consequence of the fact that the proposed approach aims at finding bugs arising from a predefined input sequence. In this way, once the uncompressed reference traces are generated starting from software, the compression ratio is evaluated as a function of k to generate the control flow checker with the best value for k to minimize memory footprint. This process is performed separately for each checker, so that the optimal value for k is computed for each FSM, depending on its intrinsic characteristics and on the specific trace that must be checked. With this procedure, it is possible to greatly reduce the memory usage on FPGA as discussed in Section 5.

5 EXPERIMENTAL RESULTS

To evaluate the methodology, an implementation has been integrated in *PandA 0.9.4* [7], an open source, publicly available framework for High-Level Synthesis developed at Politecnico di Milano. Two main changes have been applied to the framework:

(1) Efficient Path Profiling has been integrated in the generator of software executable code to generate the SW trace.
(2) The Finite State Machine generator has been extended to create the hardware checkers to be coupled with the FSMs.

The methodology flow have been verified on the *CHStone* benchmarks [17], a suite composed of 12 `C` programs, explicitly collected for representing all the possible scenarios which have to be addressed by an High-Level Synthesis tool. The benchmarks have been translated in Verilog with the default configuration of the *PandA* framework targeting a Stratix V device (5SGXEA7N2F45C1)

Benchmark	RAW	SoA	EPP	EOPT
adpcm	121553	53328	86760	40977
aes	18130	6648	2982	2406
bf	657402	203654	126658	46560
dfadd	1866	1480	598	324
dfdiv	5676	4428	3972	1194
dfmul	768	627	187	66
dfsin	193748	118629	107336	52086
gsm	25044	13508	21227	3429
jpeg	3692568	1753039	1270590	499085
mips	22904	18918	6336	6130
mpeg2	21274	12690	10443	252
sha	649508	315424	252715	51926

Table 2. The memory usage (bits) for storing the compressed execution traces.

from Intel (former Altera) with target frequency of 200MHz, using Quartus Prime Standard Edition 17.0 for synthesis.

Section 5.1 discusses the reduction of memory footprint obtained with optimized EPP, Section 5.2 shows the overhead introduced by the checkers, and Section 5.3 outlines the limits of the approach.

5.1 Memory Usage

The results of the following compression schemes were computed for each *CHStone* benchmark, to measure the effectiveness of EPP with respect to state-of-the-art FSM trace compression techniques:

RAW: no compression, i.e., traces are described by means of the list of the traversed states encoded with the smallest number of bits.

SoA: the list of the traversed states is compressed with the technique presented in [12].

EPP: the list of the traversed states is encoded with the Efficient Path Profiling as described in Sections 4.2 and 4.3.

EOPT: the execution traces are encoded with EPP and then compressed with the technique described in Section 4.6.

Note that the results obtained with the *SoA* technique depend on the encoding of the states. The states of the FSM produced by PandA are binary encoded and the states are numbered according to a depth first visit which increases the opportunities of compression of the technique used in *SoA*. In their work, the authors [12] identify the optimal number of metadata bits to use for compression, which is 6 for their implementation. Since the optimal value depends on the structure and the encoding of the FSM, the fixed number of bits identified in [12] have not been used here. Instead, for a fair comparison, the optimal size of the metadata was recomputed for every single function also for *SoA*, otherwise the results would be too biased in favor of the approach presented here.

Table 2 reports the results obtained with each technique on each benchmark. Figure 7 reports the same data normalized with respect to the non-compressed trace. The red line marks the *RAW* results, which represents the baseline.

The results show that *SoA* compresses better the benchmarks which are characterized by data dominated computation (*aes*, *blowfish*, *jpeg*, *mpeg2*, *sha*), while it does not provide significant benefits on the control dominated benchmarks (*adpcm*, *dfadd*, *dfdiv*, *dfmul*, *dfsin*, *gsm*, *mips*). Indeed, the latter type of benchmarks has Control Flow Graphs (and so also Finite State Machines) with

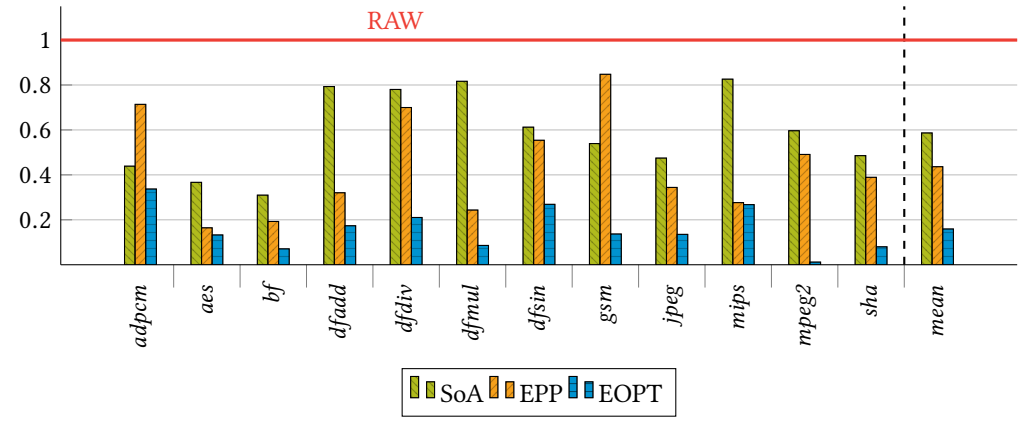

Fig. 7. Memory usage normalized with respect to *RAW*.

very large number of branches. This means that the number of transitions between states with consecutive encoding is limited, hampering the compression mechanism. On the other hand, *EPP* can compress very well data dominated traces, as well as some of the control dominated benchmarks as expected: *dfadd*, *dfmul*, *mips*. In general, the results of *EPP* are in most cases already better than the compressed state-based traces of *SoA*. However, on some control dominated benchmarks (like in *dfdiv* and *dfsin*) the benefits are only limited, and on others (like in *adpcm* and *gsm*) there are even significant penalties with respect to *SoA*. This is mainly due to the execution trace of loops without internal branches. In this type of loops all the states will be encoded with consecutive values, giving a great advantage to the compression algorithm used in *SoA*. *SoA* uses $s + m$ bits to store information about the execution of one iteration of the loop where s is the number of bits of the state encoding and m is the optimal number of bits used to store metadata. *EPP* on the contrary uses p bits to store information about the execution of the same iteration, where p is $log_2(PathMax)$ and $PathMax$ is the largest path identifier computed by Efficient Path Profiling. If the function containing the loop is characterized by a significant number of branches, the number of paths is significantly larger than the number of states, $p \gg s + m$. If one of more loops without internal branches are repeated a significant number of times, the overall size of the trace compressed by *EPP* can be larger than *SoA*. This issue arises also in compressing *dfdiv* and *dfsin*, even if their source code does not explicitly contain any loops with such characteristics. In this case indeed, the loop executed a significant number of times and which does not contain any internal conditional branch is the loop included in the Finite State Machine of the module implementing the integer division.

The results about *EOPT* show that the compression of the EPP traces described in Section 4.6 is effectively able to overcome this issue: the sizes of the compressed traces of *adpcm*, *blowfish*, *dfdiv*, *dfsin*, *gsm*, *jpeg*, *mpeg2*, and *sha* are significantly reduced with respect to *EPP*. However, the optimization introduced by *EOPT* provides benefits not only for the benchmarks characterized by loops without internal branches. If the execution trace is characterized by the consecutive repetitions of the same path inside a loop (even if it contains multiple branches), *EOPT* can further compress the execution trace. For this reason, a significant improvement is also noticeable in the compression of the traces for *dfmul*.

	#chk	Frequency (MHz)		Area (ALMs)		Power (mW)	
		nochk	chk	nochk	chk	nochk	chk
adpcm	1	220.89	213.63	5378	5716	218.22	230.14
aes	5	218.25	216.87	2238	2652	86.96	101.22
blowfish	2	214.32	221.52	1831	2040	106.30	119.26
dfadd	1	237.13	238.83	2421	2497	49.46	54.44
dfdiv	2	228.99	228.41	1879	2031	44.23	56.87
dfmul	1	219.88	219.72	904	993	32.01	36.57
dfsin	3	220.69	205.34	7990	8494	190.79	198.09
gsm	2	217.96	221.38	2316	2717	139.36	153.22
jpeg	6	214.41	209.86	9213	10264	369.47	429.61
mips	1	250.94	223.16	1016	1036	48.32	50.89
mpeg2	4	224.31	223.26	1224	1573	41.09	53.82
sha	2	256.48	221.43	1584	1820	75.22	85.78

Table 3. Synthesis results after place and route. Columns *nochk* and *chk* refer to the circuit without and with control flow checkers.

5.2 Overhead of the Tracing Logic

Table 3 reports the synthesis results obtained with Quartus Prime [11] after place-and-route phase for the accelerators produced by the PandA High-Level Synthesis flow, with and without the addition of checkers proposed in this paper. For each benchmark, the table reports the number of checkers, the obtained maximum frequency, the area overhead in terms of Adaptive Logic Modules (ALMs) and the dynamic power consumption.

The first thing to notice is that the number of checkers, reported in the column #chk, varies across benchmarks. Given that a checker is generated for every checked FSM, the actual number corresponds to the number of functions for which PandA generates a FSM and that are actually executed in the reference trace. The second interesting information is that the variation of achieved clock frequency when the checkers are integrated in the design is not significant. There are cases, like *sha* an *mips*, where it decreases of about 30 MHz, but they are also the cases where the achieved frequency without checkers was much higher than the target of 200 MHz. Despite that for these benchmarks the introduction of the checkers changes the critical paths of the circuit, the newly added paths do not introduce any timing violation since there is a significant margin between their delays and the required clock period. In other cases, like *blowfish* and *gsm*, the frequency even increases of a few MHz. In all the cases the circuits with and without checkers meet the target frequency of 200 MHz.

Results about area overhead are reported in terms of ALMs, which are the basic building blocks of the Stratix FPGA. Every ALM has 8 inputs and it consists of combinational logic and four registers. It can be configured to implement various combinations of two functions with different number of inputs, as described in [25]. The introduced overhead in terms of ALMs is correlated with the number of checkers instantiated in every benchmark. This can be inferred by the fact that *jpeg*, the benchmark with the highest number of checkers, also exhibits the largest ALM increase (+1051), while three out of the four benchmarks with only one checker (*dfmul*, *dfadd*, *mips*) are also those with the lowest ALM increase (+89, +76, +20, respectively). Clearly, these values have a different impact in percentage on each benchmark, because their sizes are very different, but the data are concordant with the fact that the main architecture of every checker is the same across benchmark. The only difference is represented by the number of bits necessary for the trace and the *increments*

memory as described in Section 4.5. This difference is also the main cause of noise in the correlation between area and number of checkers. An example of this effect is that *adpcm*, which has only one checker, shows an ALM increase of +338, while *mpeg2*, with 4 checkers only increases of +349 ALMs.

Finally, the last two columns in Table 3 report the dynamic power consumption of the benchmarks, with and without checkers. The static power dissipation is not reported because it is always equal to 1515 mW ± 2 mW across all the tests, with and without checkers. The estimation are obtained with the Quartus Prime PowerPlay Power Analyzer, after place-and-route. Given that the accuracy of the tool is ± 20% from silicon according to the documentation, it is difficult to deduce an exact equation to estimate power dissipation before synthesis. However, from the data it is possible to see that there are three main contributions to the increase of power consumption caused by the checkers: the size of the memory used for the traces, the increased switching activity that is measured when the compression is more aggressive, and again the number of the checkers. The effect of the number of the checkers and the memory used for the traces is evident in *jpeg*, which has the highest number of checkers and the largest number of memories, and shows a rise of 60.14 mW. The significance of the increased switching activity due to compression can be seen on *mpeg2*. This is the case where the *EOPT* shows the highest compression rate, and from the table it is possible to see that the dynamic power consumption increases by 12.73 mW (30%) even with the insertion of a single checker. In other cases these effects are not evident.

5.3 Limitations of the Proposed Approach

The proposed approach still presents limitations. The most evident is that it is restricted only to the control flow. While this can be important in some benchmarks, it is often not enough to pinpoint the root cause of a bug and its impact in terms of memory footprint could be negligible compared to memory for data. However, control flow plays a strategic role in debugging, because it helps to locate bugs and to give directions on which data are relevant for debugging. Compression for data traces is beyond the scope of this work, because EPP is not well suited for data compression. It certainly deserves more investigation and it could be interesting to integrate control flow with methodologies to debug single operations.

Another issue is the implicit assumption that the software testbench used for the generation of the golden reference perfectly mimics the actual behavior on FPGA. This is not necessarily true with asynchronous inputs that might affect the control flow. If these inputs occur at different times in software and in hardware, the checkers report false positives. To tackle this issue the methodology could be integrated with in-circuit assertions.

Finally, some control flow information may not be encoded directly into the FSM structure, due to architectural optimizations performed during HLS. In this case, the granularity of the checks performed by the approach is restricted to what is visible at the FSM level. It may by worth considering how to handle this kind of control flow information encoded into data. They could be handled like data as soon as a similar approach to compress data traces is developed. However, this problem is out of the scope of this work.

6 CONCLUSION AND FUTURE WORK

This work describes a methodology for improving the debug support for circuits generated with High-Level Synthesis. It is composed of two main contributions: an adaptation of a software profiling technique called Efficient Path Profiling to target Finite State Machines, and a modification of the HLS flow to generate control flow checkers that run on chip and automatically detect bugs with respect to a reference execution trace. The results reported in Section 5 show that using EPP it

is possible to greatly reduce the memory footprint of the debugging components compared to the state-of-the-art. This is possible without altering the structure of the checked FSM, with minimal frequency implications and with reasonably low area and power overhead. The whole flow can be integrated with existing techniques for automated bug detection.

The main limitations of the approach, described in Section 5.3, also represent possible directions for future improvements: investigation of compression techniques for debugging data traces; integration with in-circuit assertions to improve support for detection of timing-dependent bugs; extension of the methodology to include control flow dependent information that is not directly encoded in the FSM. These improvements would further reduce the memory usage and improve the debugging capabilities of the methodology.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

[2] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 29, Paris, France, December 2-4, 1996.* DOI: https://doi.org/10.1109/MICRO.1996.566449

[3] Nazanin Calagar, Stephen Dean Brown, and Jason Helge Anderson. 2014. Source-level debugging for FPGA high-level synthesis. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014.* DOI: https://doi.org/10.1109/FPL.2014.6927496

[4] Keith A. Campbell, Leon He, Liwei Yang, Swathi T. Gurumani, Kyle Rupnow, and Deming Chen. 2016. Debugging and verifying SoC designs through effective cross-layer hardware-software co-simulation. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016.* DOI: https://doi.org/10.1145/2897937.2898002

[5] Keith A. Campbell, David Lin, Subhasish Mitra, and Deming Chen. 2015. Hybrid quick error detection (H-QED): accelerator validation and debug using high-level synthesis principles. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015.* DOI: https://doi.org/10.1145/2744769.2753768

[6] John Curreri, Greg Stitt, and Alan D. George. 2010. High-level synthesis techniques for in-circuit assertion-based verification. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Workshop Proceedings.* DOI: https://doi.org/10.1109/IPDPSW.2010.5470747

[7] Politecnico di Milano. 2017. PandA Framework for Hardware/Software Codesign. (2017). http://panda.dei.polimi.it

[8] Pietro Fezzardi, Michele Castellana, and Fabrizio Ferrandi. 2015. Trace-based automated logical debugging for high-level synthesis generated circuits. In *33rd IEEE International Conference on Computer Design, ICCD 2015, New York City, NY, USA, October 18-21, 2015.* DOI: https://doi.org/10.1109/ICCD.2015.7357111

[9] Pietro Fezzardi and Fabrizio Ferrandi. 2016. Automated bug detection for pointers and memory accesses in High-Level Synthesis compilers. In *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016.* DOI: https://doi.org/10.1109/FPL.2016.7577369

[10] Harry D. Foster. 2015. Trends in functional verification: a 2014 industry study. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015.* DOI: https://doi.org/10.1145/2745404.2744921

[11] Intel FPGA. 2016. Quartus Prime Design Software. (2016). https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html

[12] Jeffrey Goeders and Steven J. E. Wilton. 2014. Effective FPGA debug for high-level synthesis generated circuits. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014.* DOI: https://doi.org/10.1109/FPL.2014.6927498

[13] Jeffrey Goeders and Steven J. E. Wilton. 2015. Using Dynamic Signal-Tracing to Debug Compiler-Optimized HLS Circuits on FPGAs. In *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015.* DOI: https://doi.org/10.1109/FCCM.2015.25

[14] Jeffrey Goeders and Steven J. E. Wilton. 2017. Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 36, 1 (January 2017). DOI: https://doi.org/10.1109/TCAD.2016.2565204

[15] Mentor Graphics. 2017. Catapult C High Level Synthesis, HLS Verification. (2017). https://www.mentor.com/hls-lp/catapult-high-level-synthesis/hls-verification

[16] Mohamed Ben Hammouda, Philippe Coussy, and Loïc Lagadec. 2014. A design approach to automatically synthesize ANSI-C assertions during High-Level Synthesis of hardware accelerators. In *IEEE International Symposium on Circuits and Systemss, ISCAS 2014, Melbourne, Victoria, Australia, June 1-5, 2014.* DOI: https://doi.org/10.1109/ISCAS.2014.6865091

[17] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. 2009. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *JIP* 17 (2009). DOI: https://doi.org/10.2197/ipsjjip.17.242

[18] Helen Howe. 1997. Pre- and Postsynthesis Simulation Mismatches. In *Proceedings of the 1997 IEEE International Verilog HDL Conference (IVC '97)*. DOI: https://doi.org/10.1109/IVC.1997.588528

[19] Yousef Iskander, Cameron D. Patterson, and Stephen D. Craven. 2014. High-Level Abstractions and Modular Debugging for FPGA Design Validation. *ACM Transactions on Reconfigurable Technology and Systems, (TRETS)* 7, 1 (2014). DOI: https://doi.org/10.1145/2567662

[20] Don Mills and Clifford E. Cummings. 1999. RTL Coding Styles That Yield Simulation and Synthesis Mismatches. In *SNUG (Synopsys Users Group) 1999 Proceedings*.

[21] Joshua S. Monson and Brad Hutchings. 2014. New approaches for in-system debug of behaviorally-synthesized FPGA circuits. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*. DOI: https://doi.org/10.1109/FPL.2014.6927495

[22] Joshua S. Monson and Brad Hutchings. 2015. Using shadow pointers to trace C pointer values in FPGA circuits. In *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2015, Riviera Maya, Mexico, December 7-9, 2015*. DOI: https://doi.org/10.1109/ReConFig.2015.7393364

[23] Joshua S. Monson and Brad Hutchings. 2015. Using Source-Level Transformations to Improve High-Level Synthesis Debug and Validation on FPGAs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015*. DOI: https://doi.org/10.1145/2684746.2689087

[24] NEC. 2016. CyberWorkbench: NEC's High Level Synthesis Solution. (Sept. 2016). http://www.nec.com/en/global/prod/cwb/pdf/CWB_Detailed_technical.pdf

[25] Altera Corporation (now Intel FPGA). 2016. Stratix V Device Handbook. (2016). https://www.altera.com/en_US/pdfs/literature/hb/stratix-v/stx5_core.pdf

[26] Aurélien Ribon, Bertrand Le Gal, Christophe Jégo, and Dominique Dallet. 2011. Assertion support in high-level synthesis design flow. In *2011 Forum on Specification & Design Languages, FDL 2011, Oldenburg, Germany, September 13-15, 2011*. http://ieeexplore.ieee.org/document/6069472/

[27] Konstantin Selyunin, Thang Nguyen, Ezio Bartocci, and Radu Grosu. 2016. Applying Runtime Monitoring for Automotive Electronic Development. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*. DOI: https://doi.org/10.1007/978-3-319-46982-9_30

[28] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. 1996. Identifying Loops Using DJ Graphs. *ACM Trans. Program. Lang. Syst.* 18, 6 (1996). DOI: https://doi.org/10.1145/236114.236115

[29] Andrés Takach. 2016. High-Level Synthesis: Status, Trends, and Future Directions. *IEEE Design & Test* 33, 3 (2016). DOI: https://doi.org/10.1109/MDAT.2016.2544850

[30] Liwei Yang, Swathi T. Gurumani, Deming Chen, and Kyle Rupnow. 2016. AutoSLIDE: Automatic Source-Level Instrumentation and Debugging for HLS. In *24th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2016, Washington, DC, USA, May 1-3, 2016*. DOI: https://doi.org/10.1109/FCCM.2016.38

[31] Liwei Yang, Magzhan Ikram, Swathi T. Gurumani, Suhaib A. Fahmy, Deming Chen, and Kyle Rupnow. 2015. JIT trace-based verification for high-level synthesis. In *2015 International Conference on Field Programmable Technology, FPT 2015, Queenstown, New Zealand, December 7-9, 2015*. DOI: https://doi.org/10.1109/FPT.2015.7393155