Automated Bug Detection for High-level Synthesis of Multi-threaded Irregular Applications

PIETRO FEZZARDI and FABRIZIO FERRANDI, Dipartimento di Informazione, Elettronica e Bioingegneria – Politecnico di Milano, Italy

Field Programmable Gate Arrays (FPGAs) are becoming an appealing technology in datacenters and High Performance Computing. High-Level Synthesis (HLS) of multi-threaded parallel programs is increasingly used to extract parallelism. Despite great leaps forward in HLS and related debugging methodologies, there is a lack of contributions in automated bug identification for HLS of multi-threaded programs. This work defines a methodology to automatically detect and isolate bugs in parallel circuits generated with HLS. The technique relies on hardware/software Discrepancy Analysis and exploits a pattern-matching algorithm based on Finite State Automata to compare multiple hardware and software threads. Overhead, advantages, and limitations are evaluated on designs generated with an open-source HLS compiler supporting OpenMP.

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**; • **Hardware** → **Hardware accelerators**; **High-level and register-transfer level synthesis**; *Functional verification*;

Additional Key Words and Phrases: Debugging, HLS, irregular, multi-threading, FPGA

ACM Reference format:
Pietro Fezzardi and Fabrizio Ferrandi. 2020. Automated Bug Detection for High-level Synthesis of Multi-threaded Irregular Applications. *ACM Trans. Parallel Comput.* 7, 4, Article 27 (September 2020), 26 pages.
https://doi.org/10.1145/3418086

1 INTRODUCTION

Over the past years the complexity of hardware designs has constantly increased at an always faster pace. To manage this growth and reduce time-to-market, Field Programmable Gate Arrays (FPGAs) look to be a key enabling technology for fast prototyping and design space exploration. High-Level Synthesis (HLS), using high-level programming languages for hardware synthesis, speeds up development, enabling software developers to design hardware. One of the advantages of FPGAs is their intrinsic physical parallelism that allows to speed up computations with low power consumption. For this reason, they are increasingly used in datacenters [40], High Performance Computing, and irregular applications [32]. However, compilers have limited capabilities of inferring parallelism from high-level languages, and the task of specifying parallel programs is up to the programmers. For software development, common approaches include programming languages such as OpenCL [21] and CUDA [38], the use of standard libraries such as POSIX Threads [1] (`pthreads`), or language extensions like OpenMP [3].

Authors' addresses: P. Fezzardi and F. Ferrandi, Dipartimento di Informazione, Elettronica e Bioingegneria – Politecnico di Milano, Milano, Italy; emails: {pietro.fezzardi, fabrizio.ferrandi}@polimi.it.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2329-4949/2020/09-ART27 $15.00
https://doi.org/10.1145/3418086

Following these practices, several ongoing efforts are trying to bring these programming models to maturity in HLS frameworks. In software, the support for high-level parallelism is provided by runtime libraries. For HLS the compiler is in charge to provide this support to give the illusion of the same underlying computational model.

One of the fundamental aspects for the success of HLS is the support for debugging. Despite the flourishing interest in hardware synthesis of different parallel programming languages and threading models, the methodologies for effective debugging in this area are still at their first steps. Debug support for circuits generated with HLS has received attention, but for designs synthesized from multi-threaded parallel programs the contributions are scarce. Current approaches focus on the low-level details of the infrastructure necessary for on-chip debugging [19, 47]. Users need to explicitly instruct the tools about where to place tracepoints and manually inspect the traces to spot malfunctions. As HLS frameworks grow in complexity, this can become a real burden, especially if users have little previous exposure to hardware design and to the HLS internals.

This issue is exacerbated by the fact that HLS tools are growing in complexity, steadily broadening the range of optimizations used to improve the final results. HLS is also increasingly used to generate complex systems, including third-party Intellectual Property (IP) blocks and hand-written components described with Hardware Description Languages (HDL). This means that systems must be thoroughly tested to find bugs in hand-written modules, to avoid problems due to wrong integration of components, and to rule out unforeseen faults in third-party IPs that may have not been tested in specific corner cases. Vendors usually provide testbenches and assertions along with the components, but these only help to detect that there is a bug, not to identify where the bug is, nor to pinpoint the root cause. Hence, bug identification is very time-consuming and error-prone, especially in complex systems generated with HLS, including third-party IPs and hand-written modules. Things are even worse if the original specification is a multi-threaded program and the HLS tool is trying to generate a parallel architecture for an irregular application.

This work tackles these problems, describing a technique for automated bug identification for parallel hardware designs generated from high-level specifications described with parallel programming languages. The methodology extends an existing approach, called *Discrepancy Analysis* and proposed in a previous work [14], that allows to compare the execution of the HLS-generated hardware (HW) with the software (SW) obtained from the same source code. Compared to the state-of-the-art and to the previous work [14], this article presents two main contributions: (1) a novel unified algorithm based on Finite State Automata (FSA) that allows automated comparison of hardware and software traces with a unified modular approach; (2) the extension of the FSA-based algorithm to support automated bug identification in HLS-generated multi-threaded hardware.

The proposed debug flow was integrated in an existing open-source HLS framework and evaluated on a set of benchmarks using OpenMP parallelization directives. However, the discussion is kept as general as possible to show that the same technique can be applied to debug hardware generated with other HLS tools and other parallel programming paradigms. The methodology is also agnostic about the origin of the execution traces. In this work they were collected with simulation, but the same FSA-based algorithm could be applied with data collected directly from FPGA as soon as the HLS framework provides support (see for example References [19] and [47]).

In particular, the novelty of the approach presented here is that it is able to provide at the same time all the following features: (1) trace-based fully automated bug detection without user interaction; (2) independence of the technique used for collection of the traces, which can be collected on-chip or with simulation as shown in References [14, 16]; (3) full support for parallel programming paradigms on FPGAs that does not degrade even in presence of complicated dynamic mappings between software threads and replicated hardware components. This is

the first approach that provides all these features on FPGA without being tied on a specific implementation of hardware/software thread mapping or task-scheduling mechanism.

The rest of the article is structured as follows: Section 2 introduces the motivation of our work, summarizes the state-of-the-art in the field of hardware synthesis of multi-threaded programs, and sketches a typical scenario where the methodology proposed in this article is beneficial. Section 3 recaps the necessary definitions and terminology related to traced-based Discrepancy Analysis, previously introduced in Reference [14]. Section 4 explains how to collect the execution traces necessary for debugging. Section 5 shows an algorithm for fast pattern-matching of hardware and software traces, based on Finite State Automata (FSA). This is one of the original contributions of this work, and it is the fundamental building block used in Section 6 to add support for debugging circuits generated from multi-threaded irregular applications. Section 7 discusses related works in the field of debugging methodologies for circuits generated with HLS, comparing them with the approach described in this work. Section 8 describes a proof-of-concept implementation of a debug flow using the proposed technique, showing experimental results on the kind of bugs that it can identify, and on the performance of the method. It also discusses limitations and false positives. Closing thoughts and future research opportunities are outlined in Section 9.

2 BACKGROUND AND MOTIVATION

This section describes different techniques to perform HLS of multi-threaded programs and motivates the work described in the rest of the article. Section 2.1 describes the state-of-the-art in the field of hardware synthesis of multi-threaded programs. Section 2.2 outlines the main differences between the typical use-case for parallel programming languages on FPGAs compared to HPC or Numerical Analysis due to the different hardware constraints. Section 2.3 shows a motivational example to explain some of the problems that arise in bug identification on designs generated with HLS from multi-threaded specification. This example should provide evidence of the necessity of an approach for automated bug identification and show the scenarios that it must be able to handle to be truly useful.

2.1 High-level Synthesis of Multi-threaded Programs

As FPGAs become more competitive for the acceleration of parallel workloads, the necessity for suitable programming models grows. The current trends are investing in high-level programming languages that are already industry standards for programming parallel general purpose processors, such as CPUs and GPUs, mostly based on the `C` programming language. Most of the recent HLS tools support one or more paradigms among `C` extensions such as OpenCL [21], CUDA [38] and OpenMP [3], or `C` standard libraries like POSIX Threads [1] (`pthreads`).

Efforts on OpenCL are led by the main FPGA vendors Xilinx [49] and Intel [17]. Hosseinabady and Nunez-Yanez [25] have proposed improvement to the synthesis of OpenCL workgroups in hybrid ARM-FPGA devices. Owaida et al. [39] have created a Finite State Machine with a DatAPath model suitable for execution of an OpenCL kernel, with a streaming unit to allow fast access to global data. The main work on CUDA consists of the FCUDA CUDA-to-RTL compiler (Nguyen et al. [37]) and the efforts to use it in the construction of complete System-on-Chips with the generation of the necessary interconnections, memory interfaces, and resource management components. Cabrera et al. [5] extend OpenMP directives to target more closely FPGA-specific characteristics. OpenMP loops are supported by the BAMBU compiler [9], based on GCC [43], and by the LegUp compiler [11], based on LLVM [30], which also supports nested parallelism using `pthreads`. LegUp and BAMBU exploit physical parallelism instantiating duplicated components for every thread.

Another idea is to maximize the utilization of a single hardware accelerator, extending its functionality to support hardware threads and hide latencies in pipelined loops. Halstead and Najjar extend the ROCCC HLS compiler to generate multi-threaded accelerators starting from loops constructs [22]. The programming model is similar to OpenMP for loops, and the generated architecture uses hardware context-switches to hide variable latencies due to memory accesses in irregular applications. The idea is somewhat similar to a more recent work by Tan et al. [45], but the generated architecture is different. Halstead and Najjar use deep FIFOs to realize context switch, which results in enforcing in-order termination of threads. Tan et al., instead, give an Integer Linear Programming formulation for the problem, and they explore a more general approach to avoid stalls, enabling out-of-order execution of threads in the pipeline.

There are also other works that are more focused on system integration, on Operating System supports for hybrid hardware/software threads, and on how to migrate threads to FPGA transparently in heterogeneous systems. Andrews et al. [2] define Hthreads, a multi-threaded programming model based on `pthreads`, where individual threads can be mapped to FPGA and provide the necessary runtime infrastructure in hardware and software for making this possible in a transparent way. Korinth et al. [29] instead use an OpenCL-like model. These last two works define different ways to map high-level thread directives onto instances of hardware accelerators. However, what is important to our purposes is that they are actually agnostic about this mapping and they are focused on providing the necessary system-level integration once the mapping is computed. Wang et al. [48] even consider dynamic reconfigurability for instantiating different hardware accelerators at runtime to run heterogeneous threads. The approach described in this work does not support dynamic reconfigurability. Another thing not considered here is threading support provided by means of multi-processors System-on-Chip completely placed on FPGA, as proposed, for example, by Ma et al. [31]. This kind of approach is actually not even HLS, because threads are not translated into hardware accelerators, but they run as software on the softcores on the FPGA.

2.2 Peculiarities of Parallel Programming on FPGAs

It is important to stress that parallel programming models on FPGAs are subject to very different constraints than they typically are in software.

In software, parallel programming models are typically used to squeeze the most performance out of high-end machines. Having hardware constraints is not a common problem. It is far more common to be forced to optimize the software to saturate the performance limits of the hardware.

On FPGAs, resources are limited. Complex program logic is translated into hardware components that control the design execution, and complex operations are translated into combinatorial datapaths that perform the computation in parallel. All these components are mapped onto configurable blocks, and memory is fragmented in small RAM blocks interspersed with the FPGA fabric. In this sense, the most scarce resource on FPGA is area, and everything consumes area: program logic, data processing, memory.

This means that the typical design that is implemented on FPGAs using these programming models is much smaller than the typical parallel program executed on CPUs.

An interesting consequence of these area constraints is that they restrict the class of algorithms that can be efficiently accelerated on FPGA. In practice, they tend to be hot loops of parallel programs. These, by their nature, are very homogeneous loops, in the sense that all the iterations perform the same computations on different data. The performance-oriented way to map these loops on FPGAs is to physically clone the body of the loop in hardware, so loop multiplicity is translated into physical parallelism. In this scenario, parallel programming directives (such as OpenMP `#pragmas`) are used to instruct the compiler to parallelize the loops.

Given that we propose an FPGA-oriented approach, we will focus our attention on parallel programs with these properties. Automated bug detection for parallel program on FPGAs is still an emerging topic, so we do not strive to cover the full expressive power of the mentioned programming models. Such models have been designed and shipped across several years, have a much wider scope and richer semantics, which not always has a counterpart on FPGA. This article focuses on improving the current status of debugging techniques on FPGA for such languages.

2.3 Motivating Example

This article proposes an approach for automated bug detection, based on comparison of multi-threaded hardware and software executions. The main challenge to do this is to find a unified way to resolve the mapping of software threads onto hardware for the variety of methodologies described in Section 2.1.

In general, HLS of multi-threaded programs needs to consider the following issues:

Task homogeneity – i.e., if all tasks execute the same function or not. Among the mentioned programming models, `pthreads` is the only one able to provide heterogeneous tasks. In HLS, each thread that executes a separate task is mapped onto a dedicated hardware component. Nowadays, HLS tools that support `pthreads` only support specifying this mapping ahead of time. In practice, these functions execute in parallel, but they do not pose particular challenges to debugging, because each module is only associated to a single high-level function. For this reason, in this article, we ignore this model, which can already be treated with available debugging techniques.

Hard or Soft Threads – i.e., if threads are implemented with duplicated hardware components or with dedicated additional logic to suspend and resume multiple logical threads on the same accelerator to hide memory latencies. Various HLS techniques mentioned in Section 2.1 use both hard or soft threads. Context switching of soft threads complicates the mapping between hardware and software executions, so our methodology needs to handle it.

Static or dynamic dispatch – i.e., whether the assignment of a task to a certain hardware accelerator is decided statically at compile time or dynamically during the execution. Again, there are some techniques that provide dynamic dispatching of tasks onto hardware components to compensate imbalance dynamically and reduce stalls [10]. Dynamic dispatching complicates the mapping between software and hardware threads, so our approach needs to support also this scenario.

The remainder of this article focuses on the homogeneous parallelism offered by OpenMP, CUDA, and OpenCL, where dynamic dispatch, hardware context-switches, and variable latencies typical of irregular applications complicate the mapping of software tasks onto hardware accelerators, and every task is dynamically associated with some form of task identifier (OpenMP loop iteration variable, CUDA thread ID, and OpenCL work-item).

From now on, we will adopt OpenMP notation for the examples, but the discussion is valid for all the other homogeneous parallel programming paradigms.

Consider for example the source code in Figure 1, where function `f` contains an OpenMP for loop with memory accesses and multiple calls to the function `g` without data dependencies. Without support for multi-threading, a typical HLS tool would generate a component for `f` and one for `g`, instantiating the second in the first. If the alias analysis can tell that the memory accesses on `A[i]` and `B[i]` are on disjoint memory locations, it could duplicate the instances of `g` and execute them in parallel inside the loop body. Otherwise, it would generate a single instance of `g` and serialize the calls inside the loop. This is clearly suboptimal for performance.

```
int A[8], B[8]; int g(int x);
int f(int a, int b) {
    #pragma omp parallel for reduction(+:a,b)
    for (int i = 0; i < 8; i++) {
        a += g(A[i]);
        b += g(B[i]);
    }
    return a - b;
}
```

Fig. 1. C function with an OpenMP for loop. HLS can generate different multi-threaded architectures.

If the HLS has support for OpenMP, instead, there are a number of possible optimizations. The loop body could be treated as a separate module and physically replicated multiple times, resulting in physical parallelism. This is what is done by LegUp [11] and BAMBU [9]. Otherwise, a parallel architecture could be generated for the loop body, allowing simultaneous executions of different threads at the same time, exploiting context switching to hide memory latencies due to the accesses to `A[]` and `B[]` in the loop body. This is what is done by the CHAT compiler [22] based on ROCCC and by Tan et al. [45]. The two techniques could even be used together, physically duplicating the loop body, and enabling context-switches on all the duplicate copies, depending on some design space exploration trade-offs. This gives an idea of the complexity that an approach for automated bug detection must be able to handle. The most general assumption is that it must be possible to support physical and hardware-thread parallelism with dynamic scheduling of tasks onto accelerators.

In such a scenario, the Discrepancy Analysis proposed in Reference [14] would not be able to tell which execution of what hardware accelerator has to be compared with a given software thread to understand if the hardware is behaving correctly or not. Other similar approaches for automated bug detection do not consider the problem of starting HLS from multi-threaded specifications [28], as discussed in Section 7.

However, these programming models are increasingly used and it is unrealistic to delegate to users the burden of unraveling the complexity of HW/SW thread mapping when the generated design contains bugs. Bugs could be located in third-party components integrated in the design with HLS, which may not be suitable for some parallel execution (especially if context-switch is involved). In this case the end-users need some way to know it without having to understand all the optimizations performed by the HLS tool in the generation of the design. In theory, some bugs could even be introduced by the HLS tool itself, either because the original code cannot actually be parallelized safely or because some of the optimizations performed by the tool make some assumption that are not true in users' code. Hence, HLS tools need to provide an environment for automated bug identification to solve this problem, improve the design experience, and reduce the time necessary for debugging.

3 FUNDAMENTALS OF DISCREPANCY ANALYSIS

This section introduces some of the fundamental concepts of Discrepancy Analysis, mostly adopted from Reference [14]. Discrepancy Analysis is founded on the notions of *Execution Traces* and of equivalence between SW and HW executions.

Execution Traces are lists of values that can be collected from HW and SW executions with different techniques and that describe such executions at two levels: *control flow level* and *data level* (also called *operation level* in the following and in Reference [14]). These traces can be understood in terms of general high-level representations used in HLS compilers: *Control Flow Graph* (CFG), *Finite State Machine* (FSM), and *DataPath*.

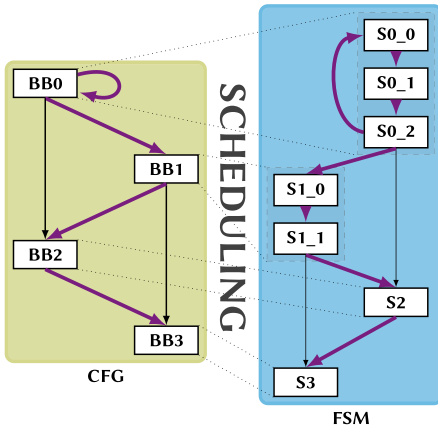


Fig. 2. Scheduling relationship between Control Flow Graph and Finite State Machine. Purple thick arrows show the hardware and software executions.

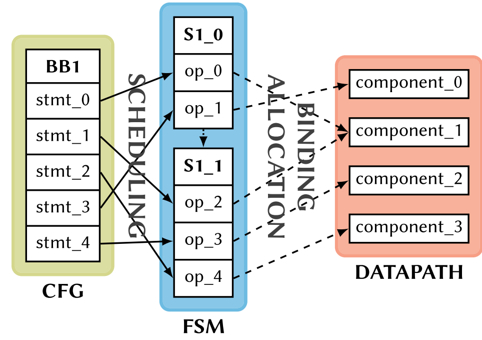


Fig. 3. Scheduling, binding, and allocation maps between statements in Basic Block, operations in Finite State Machine, and DataPath components.

The CFG is a directed graph representing the control flow of the original program. It is built by the compiler frontend, and its nodes are called *Basic Blocks* (BB). Every BB contains a sequence of consecutive operations. Every Basic Block also has a single entry point at the beginning and one or more branching conditions at the end.

The FSM and the DataPath, instead, describe the generated HW and they are created by HLS tools starting from the CFG after frontend optimizations. The creation typically requires three tightly related steps: scheduling, allocation, and binding. The whole process can also involve non-trivial modifications, such as sharing, chaining, pipelining, and duplication of operations in more than one state. For a given CFG, the scheduling decides, for every operation in a BB, the state of the derived FSM where it will be executed. What is important for Discrepancy Analysis is that the HLS engine creates the FSM from the CFG in such a way that every BB is mapped onto a chain of states in the FSM, with branches only at the end of the chain.

3.1 Control Flow Level

Consider a function `f` described in a high-level language such as `C` and its Control Flow Graph after frontend compiler optimizations. From such a graph, HLS produces an FSM and a DataPath. For considerations on control flow, only the FSM is necessary. The FSM itself can be represented as a graph, like in Figure 2. With the appropriate conventions, CFG and FSM can accept the same inputs. The CFG represents the execution of the software and the FSM the execution of the generated HW. The two flows have different semantics for operations: sequential in BBs; concurrent or chained in a state of the FSM. However, from a control flow standpoint, the execution can be described as an ordered list of nodes visited on the graph, being it BBs for CFG or states in FSM.

Definition 3.1. The *Software Control Flow Trace* (SCFT) on a given input I is the ordered sequence of BBs representing the execution of the CFG.

Definition 3.2. We call *Hardware Control Flow Trace* (HCFT) on the input I the ordered sequence of states describing the execution of the FSM.

In the following, SCFTs and HCFTs together are called with the general term *Control Flow Traces* (CFT). According to the definitions, the CFG can be regarded as a function S_{cf} that associates a

Software Control Flow Trace $S_{cf}(I)$ to every input I. In Figure 2 the SCFT is ⟨ **BB0, BB0, BB1, BB2, BB3** ⟩. Similarly, the FSM can be considered as a function H_{cf} that associates a Hardware Control Flow Trace to every input I. In Figure 2 the HCFT is ⟨ **S0_0, S0_1, S0_2, S0_0, S0_1, S0_2, S1_0, S1_1, S2, S3** ⟩. Using these concepts, it is possible to define equivalence at control flow level.

Definition 3.3 (Equivalence of Control Flow Traces). Let be fixed an input I for both a CFG and its associated FSM. Let then be $S_{cf}(I) = \langle BB_0, BB_{k1}, BB_{k2}, \ldots, BB_{K(I)} \rangle$ and $H_{cf}(I) = \langle S_0, S_{j1}, S_{j2}, \ldots, S_{J(I)} \rangle$ the related Software and Hardware Control Flow Traces. $S_{cf}(I)$ is *equivalent* to $H_{cf}(I)$ if $H_{cf}(I)$ can be produced from $S_{cf}(I)$ substituting (BB_k) with the states associated to it by scheduling.

3.2 Data Level

Discrepancy Analysis at the control flow level cannot find bugs that do not alter the control flow and cannot locate their root cause even if they affect control flow. To overcome this limitation it is necessary to refine the granularity at the data level, considering also HLS information from binding and allocation. This means integrating information from the FSM and from the DataPath.

Figure 3 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 components in the DataPath. Note that the mapping of operations on HW components is many-to-one, meaning that components can be shared by multiple operations if their execution does not overlap. Instead, there is a one-to-one mapping between the statements in a BB and all operations scheduled in the related states.

The fundamental assumption for the definition of execution traces at the data level is that every statement cannot be scheduled twice in a chain of states representing a single BB. In hardware synthesis it is common practice to speculate an operation and to schedule it in more than one state in the FSM to reduce the execution cycles. However, the states where it is scheduled must be distinguishable from a control flow standpoint. Another way to state this is that all the duplicated copies must be executed under different conditions, even with speculation and guard conditions. If this was not true (so that two copies were executed under the same conditions) the behavior would not be consistent with the high-level specification, where the operation was executed only once.

With this assumption it is possible to define execution traces at the data level: *OpTraces* (OT).

Definition 3.4. Let O_i be an operation in a Basic Block. The *Software OpTrace* (SOT) of O_i is the list of the results $s_{1,i}, \ldots, s_{k(i),i}$ of the operation across all the execution.

Definition 3.5. Let O_i be the same operation scheduled in a state $S(O_i)$ of the associated FSM. Let also $C(O_i)$ the component in the DataPath that was allocated and bounded in HLS to execute operation O_i. The *Hardware OpTrace* (HOT) of O_i is the list of values of the output signal(s) of $C(O_i)$ collected during hardware execution when the FSM was in state $S(O_i)$.

Notice that OpTraces were defined per Basic Blocks in Reference [14], but for the purposes of this work the definition is formulated with a finer granularity: one OpTrace for every single operation. The definition is consistent with Reference [14], because a BB is just a list of operations. Hence, what is called OpTrace in Reference [14] for a Basic Block is just a set of OpTraces as defined here for a single operation.

Definition 3.6 (Equivalence of OpTraces). Let O_i be an operation in a Basic Block $BB(O_i)$ of a CFG. Consider a Finite State Machine constructed from the CFG during the HLS process and call $S(O_i)$ the state where O_i is scheduled. Let also $C(O_i)$ the component in the DataPath that was allocated

and bounded in HLS to execute operation O_i. The Software OpTrace $S_{op}(O_i)$ and the Hardware OpTrace $H_{op}(O_i)$ are *equivalent* if they are equal through some equality function.

Notice that the equality function can be as simple as bitwise equality for plain integer data, but it can be complicated in case of floating points or custom data formats, up to involving context-dependent address translation tables for pointers and addresses [15].

3.3 Equivalence of Execution Traces

Equivalence between hardware and software is defined in terms of equivalence at the two levels.

Definition 3.7 (Software Traces). A *Software Trace* (ST) for a CFG on a given input I is a pair $S(I) = [S_{cf}(I), S_{op}(I)]$, where $S_{op}(I)$ is the set of Software OpTraces $S_{op}(O_i)$ such that O_i is an operation in one of the Basic Blocks in $S_{cf}(I)$.

Definition 3.8 (Hardware Traces). A *Hardware Trace* (HT) for an FSM on a given input I is a pair $H(I) = [H_{cf}(I), H_{op}(I)]$, where $H_{op}(I)$ is the set of Hardware OpTraces $S_{op}(O_i)$ such that O_i is an operation scheduled in one of the states in $H_{cf}(I)$.

Definition 3.9. Let $S(I) = [S_{cf}(I), S_{op}(I)]$ be a Software Trace for a Control Data Flow Graph and $H(I) = [H_{cf}(I), H_{op}(I)]$ be the Hardware Trace of the associated Finite State Machine generated during HLS. $S(I)$ and $H(i)$ are *equivalent* if both the following conditions are satisfied:

(1) *equivalence at the control flow level* — $S_{cf}(I)$ is equivalent to $H_{cf}(I)$ according to Definition 3.3.
(2) *equivalence at the operation level* — there is a bijective relationship $\equiv$ between $S_{op}(I)$ and $H_{op}(I)$ such that $S_{op}(O_i) \in S_{op}(I) \equiv H_{op}(O_j) \in H_{op}(I) \iff i == j$ and $S_{op}(O_i)$ is equivalent to $H_{op}(O_j)$ according to Definition 3.6.

The key insight of these definitions is that Control Flow and Operation levels describe the two different ways in which the hardware execution can differ from the original specification. At the operation level, all the assignments are checked. The only constructs that are not checked directly at the operation level are control flow instructions (branches, function calls, and return statements) that are handled at the control flow level. For branches, the evaluation of the branching condition is checked at the operation level. The CFTs are used to verify that the correct branch has been taken. Function calls and return statements are treated similarly. Calls are scheduled in states of the FSM, so the control flow is enough to check their start and end. OpTraces are involved as well, since they are used to check that the passed parameters match those used in software. Return statements, even if they are not checked directly in the returning function, are checked in the OpTrace of the caller. In this way all the instructions are either directly checked at the operation level or indirectly checked using both control flow and data information.

4 GENERATION AND COLLECTION OF THE TRACES

The definitions of Section 3 do not explain how to extract the traces from hardware and software execution. This is good, because they do not depend on the specific method used to generate and collect the traces, making them suitable to use with simulation-based methodologies as well as with traces directly collected from FPGA trace buffers. In this section, we describe the methods used in our proof-of-concept to generate and collect the traces, along with a method for automatic identification of the necessary signals in the generated hardware. This is necessary to understand the rest of the discussion and the main contributions of this work.

Software Traces. The high-level code is instrumented to generate the Software Traces when executed. Instrumentations are added directly in the IR to have a finer granularity and control on

compiler temporary variables introduced for optimizations. Then the IR with instrumentations is printed back in C. The code generator is designed to structure the instrumented code like the CFG. It starts from the IR of the compiler in Static Single Assignment form (SSA [13]), and it prints it back in C, splitting SSA's ϕ operations as described in Reference [4]. In this way all the operations that are not control flow instructions can be printed as assignments. In SSA, every variable is assigned only once, and every printed statement assigns only one variable. In this way, to generate the SOTs it is enough to print the value of every variable (which has a unique identifier) after its assignment in SW. To generate the SCFTs a print instruction is placed at the beginning of every BB to dump the identifier of the BB itself each time it is executed. Concretely, the obtained SCFTs are lists of the identifiers of the BBs traversed at runtime. Similarly, SOTs are the lists of results of assignment statements. None of the software traces contains timing information.

Hardware Traces. To generate Hardware Traces, the relevant signals must be identified in the design. This is entirely possible without user interaction, because all the necessary information is already available in the HLS engine. First, the clock source of the design must be extracted. This is necessary to drive the whole comparison, to understand the timing and the duration of all the other signal variations. This signal is named `clock` in the following. Hardware Control Flow Traces are lists of states traversed by the FSM during execution. Hence, the signal used to produce them is basically the state signal of the FSM. It is denoted as `state` in the remainder of this work. Handling the execution of a function typically requires two other signals: one, asserted by the caller, to start the execution; the other, asserted by the called function, to notify the caller that the execution ended. The signals involved in this handshaking mechanism are called `start` and `done`, respectively. Usually every functional module stays in its initial state when it is not executed. Then it is necessary to check for `start` and `done` to have the full information on the execution. The necessary signals to produce HCFTs are `state`, `start`, and `done`, for all the synthesized functions.

For OpTraces the identification of the signals relies heavily on the binding information coming from HLS. According to the definitions in Section 3, HOTs are composed by the values of the output signals of the HW components in the DataPath used to implement the operations in the FSM, which in turn are associated with operations in the CFG. But these things are part of what is computed in HLS during binding and allocation. The details of the signal naming are strictly implementation-dependent and change from an HLS tool to another, but every HLS compiler must know this particular piece of information. The only additional signals to be traced are the `start` and `done` signals used for the handshaking mechanism of Variable Latency Operations (VLO).

Once all the necessary signals have been detected in the design, it is possible to generate the Hardware Traces. The proof-of-concept described in this article relies on simulation, because it is the easiest way to provide full observability on the selected signals and registers without altering the design. However, not all the signals must be traced during the execution, as explained above. This allows to reduce the volume of the traces to make them more manageable even for larger designs and longer simulations. Moreover, the approach described in this work could be applied to traces directly collected on-chip (like for example in Reference [16]), as long as it is possible to provide observability on the necessary signals. With simulation, the design is executed with the same input as the C program, and the signal variations are dumped in compressed Value Change Dump format (VCD). The necessary signals are just a small portion of the total and are selected automatically, reducing the VCD only to what is really needed for the Discrepancy Analysis. This yields a considerable reduction of the VCD size, with obvious benefits for I/O time.

5 COMPARING EXECUTION TRACES USING FINITE STATES AUTOMATA

This section describes how to compare the traces to check for equivalence. The comparison can be performed separately on control flow and data level, using a unified pattern-matching

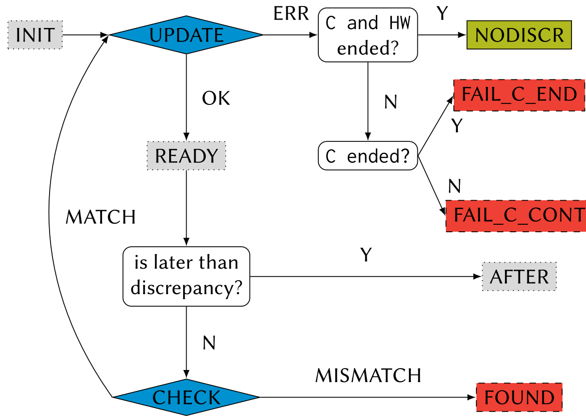


Fig. 4. FSA for the comparison of the traces.

algorithm. Separating the two levels makes the algorithm easier to understand, while using the same algorithmic template makes the whole approach mode modular and allows to easily extend it to multi-threaded code in Section 6. The method used here is based on Finite State Automaton (FSA), and it represents one of the novelties of this work compared to Reference [14]. Please bear in mind that the term Finite State Automaton is purposely used to avoid confusion with the Finite State Machine of the hardware controller. In the remainder of the article, the word *status* is always referred to the FSA, while the word *state* always refers to the FSM.

5.1 Finite State Automaton for the Comparison of the Traces

Despite the fundamental semantic difference between control flow and data traces, the basic algorithm for their comparison can be based upon a Finite State Automaton with the same structure. An FSA of this kind works on a pair of associated traces: one for hardware and one for software, and is depicted in Figure 4. The possible statuses of the FSA are represented by rectangular nodes: INIT, READY, NODISCR, FAIL_C_END, FAIL_C_CONT, AFTER, and FOUND. There are three kinds of statuses represented by different type of nodes: (1) gray with dotted borders – the FSA has not yet checked the next entry in the traces looking for a discrepancy; (2) green – success (only NODISCR), the FSA completed the analysis of the traces and no mismatches were found; (3) red with dashed borders – ending state representing failures, i.e., a discrepancy was detected. Among this last group, FOUND is when an actual mismatch between Hardware and Software Traces is actively detected. FAIL_C_CONT is when the Software Trace continues even if the Hardware terminates and FAIL_C_END is when `C` ends prematurely, while HW keeps going. In the figure, the blue diamond-shaped nodes are functions that manipulate the traces. These are the only parts of the automaton that operate differently for control flow and data (see Section 5.2).

For every couple of hardware and software traces the FSA starts in status INIT and operates as follows: UPDATE slides through the traces, selecting the next value available in the software trace. It then uses them with HLS information to compute the next relevant time when the hardware trace must be compared with software. Remember that the software trace is untimed, while the hardware trace has timing information. UPDATE returns ERR when some of the data necessary for the evaluation of the mismatch cannot be computed. In this case the kind of error is determined and the FSA terminates. Otherwise, the FSA becomes READY. If the timing of the next entry in the hardware traces is later than another discrepancy previously detected by another automaton for another couple of traces, the FSA suspends the checks, entering in AFTER. Indeed, if a discrepancy

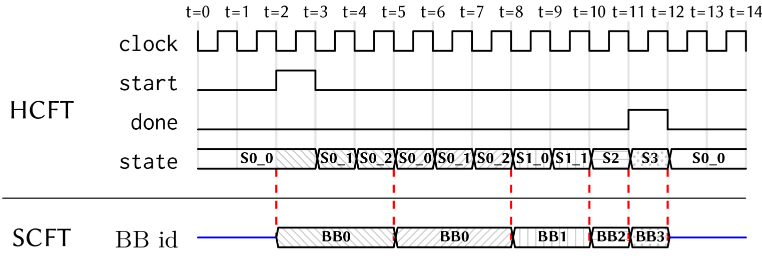


Fig. 5. Relationship between Control Flow Traces. The HCFT is represented by the first four signals, while the SCFT is the list of Basic Block identifiers. The traces are referred to the CFG and FSM shown in Figure 2. Dashed lines between `state` and `BB ID` represent the scheduling relationship between states and basic blocks.

is detected on an operation, the following are likely affected, so only the first discrepancy is important. Skipping checks on traces with higher timestamps makes the comparison faster. If no prior discrepancy was found, the CHECK function tells if the two traces actually match. If they do, the cycle restarts with the next entries, otherwise the FSA enters FOUND and terminates.

5.2 Algorithms for Comparison of the Traces

Here, we detail two high-level algorithms for trace comparison, explaining how the UPDATE and CHECK functions in the FSA operate for Control Flow Traces and OpTraces, respectively.

Control Flow. The comparison of Control Flow Traces is performed one function at a time. The HCFT for a single function consists of four signals: `clock`, `start`, done, and `state`. The SCFT for the same function is simply a list of Basic Block identifiers. An example is shown in Figure 5 on traces referred to Figure 2. From the figure, it is straightforward to understand how the CFTs can be compared. In this case the UPDATE and CHECK functions can be coalesced in a single one, which operates in the following manner: First, it considers the next BB ID in the SCFT, and it uses the scheduling map computed during HLS to obtain the list of states in the FSM associated to that basic block. This mapping is depicted with a red dashed arrow in the figure. Finally it checks that the `state` signal in the hardware trace are coherent with the identifiers computed from scheduling. The `clock`, `start`, and done signals are used to ensure that the FSM is actually in execution.

Operations. The analysis of the OpTraces is performed one operation at a time. Figure 6 shows an example of CFG and FSM with the traces related to two operations (op1 and op2). These are the data manipulated by the FSA for the comparison of OpTraces. The comparison is depicted in Algorithm 1.

It works on HW and SW traces, and it fills a map of discrepancy reports for every operation. The main loop works on a single variable at a time, selecting HW and SW traces and passing it to the FSA described in Section 5.1. In line 6 of Algorithm 1 the FSA is called as if it was a function. The meaning is that the FSA associated to the traces is executed on the traces up to termination. The result of the execution of the FSA on the traces of an operation is a terminating status cur_status, representing information on the discrepancies for that operation. At the end of the analysis of all the operations, if even a single element in discr_status_map reports a mismatch the bug is reported to the user.

The UPDATE function updates the traces using different strategies for Fixed Latency Operations (FLO) and Variable Latency Operations (VLO). FLOs can be simple operations, chained operations, and also pipelined modules. Their execution time is fixed, known at compile time, and used by the scheduling algorithm to decide how to structure the FSM. VLOs are typically used to model function calls, external memory accesses, or operations with long execution times. Long operations

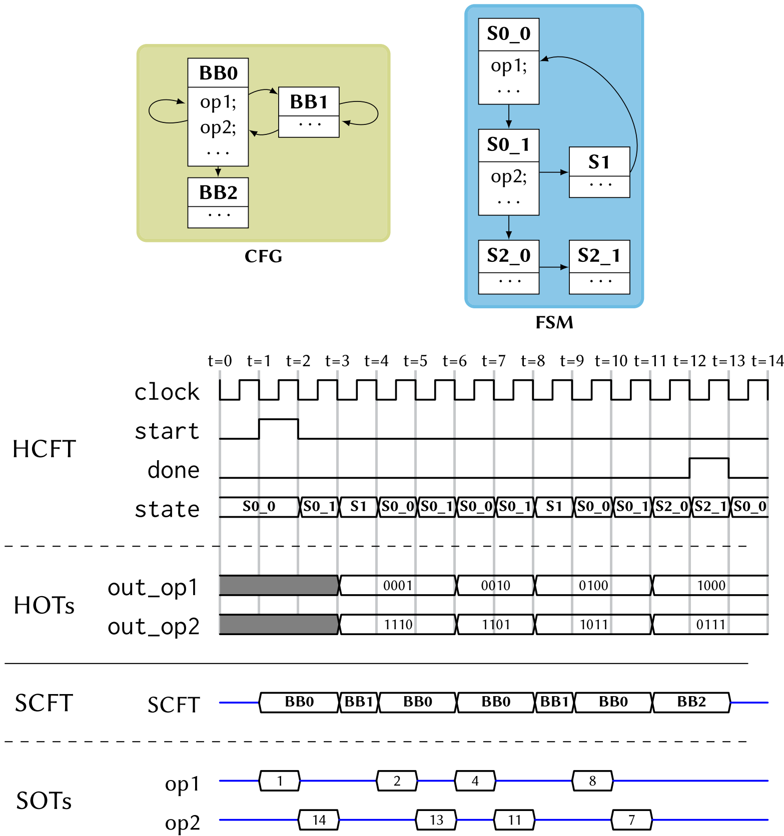


Fig. 6. Visualization of Hardware and Software Traces, with the CFG and FSM used to generate them.

could be treated FLOs but, unless there are plenty of other operations without data dependencies, it would require several waiting states, increasing the area of the FSM. For VLOs the execution time is assumed to be unknown, so they are handled with a handshaking mechanism involving a `start` and a done signal, which are part of the OpTraces for these kind of operations.

The UPDATE function is described in Algorithm 2. It flows through the SOT to get the next assigned value in C (line 1) and through the HCFT to get the start_time of the new HW execution (line 2). Lines 3 to 5 perform sanity checks on the new start time and on the SOT. If the SOT is empty there is nothing to compare the HW execution with. Moreover, if the detection of the start_time fails, it means that the HCFT of the FSM never enters in a starting state for the operation again. This means that the operation is executed in C but not in HW, so it is marked as an error. From lines 6 to 13, the end_time in HW of the newly started operation is computed. This is needed to compare the output signal with the SW value only after the operation is complete. For FLOs, execution time is fixed, so it is simply added to start_time (lines 6–8). For VLOs the done port must be checked (lines 9–11). If it is never asserted, UPDATE returns ERR, otherwise OK.

The algorithm on its own may not be enough to understand how the checker FSA works in practice. A simple example can be demonstrated with the traces sketched in Figure 6. Consider op1. It is in **BB0** and it is scheduled in **S0_0**, so **S0_0** is a starting state for op1. Assume it is a FLO with execution time of two cycles. The FSA starts in state INIT. It then runs the UPDATE function. The SOT of op1 is not empty, and its first value is 1. The start_time is computed looking

ALGORITHM 1: Discrepancy Analysis for OpTraces

```
Input: Hardware and Software Traces
Output: discr_status_map
 1: discr_status_map[] = empty;
 2: for all (O_i operations in the program) do
 3:    select the following:
       f – the function where O_i belongs
       C_H – Hardware Control Flow Trace for f
       C_S – Software Control Flow Trace for f
       O_H – Hardware OpTrace for O_i
       O_S – Software OpTrace for O_i
 4:    cur_status = NODISCR;
 5:    repeat
 6:       cur_status= FSA(C_H, O_H, C_S, O_S);
 7:    until (cur_status != NODISCR and
       cur_status != FOUND and
       cur_status != FAIL_C_END and
       cur_status != FAIL_C_CONT and
       cur_status != AFTER)
 8:    discr_status_map[v_i] = cur_status;
 9: end for
```

ALGORITHM 2: Pseudocode for the UPDATE function

```
Input: Same as the FSA
Output: OK if ready for next comparison, ERR otherwise
 1: select next value in SOT;
 2: start_time = time of the next starting state for operation;
 3: if (no starting state was found or SOT is empty) then
 4:    return ERR;
 5: end if
 6: if (is FLO) then
 7:    end_time = start_time + exec_time;
 8: else
 9:    end_time = (first time after start_time when done = 1);
10:    if (done is never asserted) then
11:       return ERR;
12:    end if
13: end if
14: return OK;
```

at the HCFT. **S0_0** starts at t=0, but given that it is the initial state, the real computed start_time is t=1. Adding the execution time the end_time results 3. Then the value of `out_op1` is checked at time t=3. The binary value (0001) is is compared with the SOT, using the CHECK function. In this case the comparison is straightforward and MATCH is returned. Then the UPDATE function is called again, iterating this process other three times to check all the four assignments. The fourth time the UPDATE function is called, it returns ERR, since the SOT is empty and there are no new

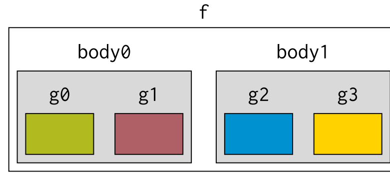


Fig. 7. Structural layout of the parallel microarchitecture generated from the C code in Figure 1.

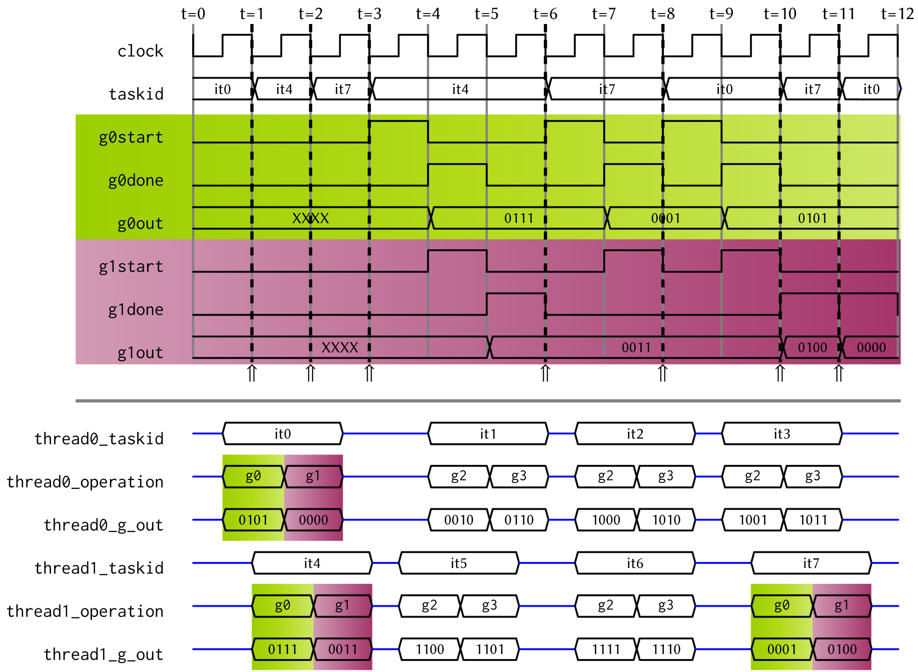
starting states in the HCFT. The FSA enters the NODISCR status, and the analysis of this trace ends. The same operations are performed on all the other traces to completion. If at a certain point in hardware execution one of the operations gives a wrong result, the comparison with software will fail, detecting a bug. For example if out_op2 at t=3 was (1,010) instead of (1,110) the algorithm would have detected a mismatch. The same holds if one of the hardware traces ends earlier than software or vice versa.

6 DEBUGGING CIRCUITS GENERATED FROM MULTI-THREADED PROGRAMS

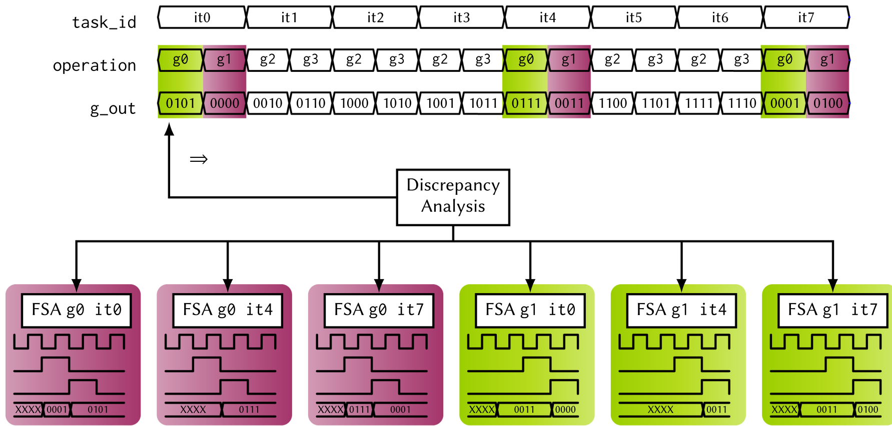
The Discrepancy Analysis described in Section 5 does not support multi-threading, actually not even procedure cloning [46]. The reason is that Sections 3, 4, and 5, always make the assumption that there is a one-to-one mapping between software and hardware traces. This is equivalent to the assumption that there is always only one hardware accelerator for every high-level function and that every accelerator only executes one task to its completion before starting a new one. But this is not what happens when HLS starts from parallel programming directives.

Consider again the motivational example reported in Section 2.3 in Figure 1, and suppose that the HLS tool instantiates two physical copies of the loop body (body0 and body1) like in Figure 7. Suppose then that the memory accesses are recognized as separate memory locations and the tool also instantiates two physical copies of g inside every copy of the loop body. In addition, suppose that the assignment of iterations of the loop onto body0 and body1 is not statically assigned but decided at runtime from some kind of component implementing dispatch policy. Finally, suppose that both the copies of the loop body support dynamic context switching, so they can request another iteration to the dispatcher if the currently executed iteration stalls waiting for variable memory latencies on accesses to A[] or B[]. For example, it could happen that iterations 0, 4, and 7 of the loop are assigned to body0, while iterations 1, 2, 3, 5, and 6 are assigned to body1. Let us focus on body0. With dynamic scheduling and context switch it may be possible that iteration 0 starts, it stalls on memory request for A[i] and B[i], and it is context switched to yield the DataPath to iteration 4. The same stalls happen then for iterations 4 and 7, for example, because the memory for A[] and B[] is off-chip and has irregular latency. For the same reason it may happen that the requests of these iterations are served out-of-order and that the context switch logic decides to wake up the three iterations in reverse order to mask latencies. An example of this reordering is shown in Figure 8(a).

The top portion of the figure represents the hardware. The dashed vertical lines marked by the small arrows on the bottom (t=1, 2, 3, 6, 8, 10, and 11), are the instant where body0 performs a context switch. The taskid represents the iteration in execution at any given moment. The other six lines, grouped in two blocks of three with the same background, represent the start, done, and output signal of each instance of g inside body0. We can see that the executions of g0 g1 in the same iteration can overlap, as in g0 at time t=4, where g1 starts even if g0 has not yet finished. Also, the execution of g0 and g1 can be suspended if the iteration is context switched, and they are resumed later when the proper iteration returns in execution. As an example of this pattern, see the call to g1 of iteration 7 is started at t=7, suspended at t=8 and finally terminated at t=10.



(a) Execution traces as they are obtained directly from software in Fig. 1 and hardware in Fig. 7.



(c) Preprocessed execution traces ready for Discrepancy Analysis.

Fig. 8. Visualization of the comparison of the traces generated by the code of Figure 1 and the circuit of Figure 7.

The lower part of Figure 8(a) portrays what happens in software. In this part there is no timing information, the actual number of threads in execution is different, and the assignment of tasks to threads is not the same as in hardware. With all these differences, the key information to perform the Discrepancy Analysis is the task ID (or iteration ID). In software, it is the iteration that is executed by a certain thread at a given moment and it can be dumped during execution with additional dedicated instrumentation. In hardware, it is the task currently executed by

an accelerator and it can be extracted with an appropriate signal selection guided by HLS, by inspecting the components that manage the assignment of the tasks and the context switch. Using this information, the traces are preprocessed before the Discrepancy Analysis. The preprocessing is different for hardware and software. The software traces are merged and filtered according to the sequence of task ids, as shown in Figure 8(c). In this way it is possible to obtain a single trace from all the traces scattered across the different software threads. The hardware traces are filtered, again using the task ID. In this phase if there are some task IDs that are executed in software but not in hardware it is already possible to detect a bug. If all the task IDs executed in software are also executed in hardware, instead, a single hardware trace is extracted for every executed task ID. The result, shown in Figure 8(c), is that we have a single software trace to be compared with a set of hardware traces. Hardware traces have lost part of the timing information, but they maintain consistency of the internal ordering. They just happen to have "jumps forward" in time, when the task was suspended and another one was in execution on the accelerator. With this setup, the algorithm described in Section 5 can be adapted, instantiating a separate FSA for every task ID. This FSA works only on its filtered vision of the hardware trace, but the inner functioning is exactly as described in Section 5. The comparison starts from the software trace, looking for the iteration ID and using it to decide which FSA has to handle the next comparison. Given that the FSAs are stateful, it is not a problem if the trace associated to a certain task ID is not checked consecutively from beginning to end. When the software trace ends, the analysis reports the detected errors as well as if there are still some values in the hardware traces to be checked, meaning that the hardware has executed more operations than the software.

7 RELATED WORK

There are several different ongoing efforts to endow HLS frameworks with effective support for debugging, both in academia and in industry [20, 36, 44]. The approaches are very varied and focus on different aspects of the infrastructure necessary for debugging, but very few currently take into consideration code generated from high-level threading directives.

Some approaches analyze architectures and compiler support for automatic and efficient generation of components for on-chip debugging. Monson and Hutchings [34] use source-level transformations for the insertion of tracing logic (Event Observability Port and Buffers) for the output signals of operations, but they do not consider multi-threaded code. Goeders and Wilton generate dedicated debugging components, reducing the memory footprint of the traces on FPGA, handling compiler temporary variables [18, 28] and multi-threaded code [19]. Their main goal is to provide a software-like debug framework, where users can inspect the traces after execution or suspend the hardware to analyze its state. They do not provide automated bug detection, unlike what is proposed in this work. They also note that suspending the execution may break interactions with other components of the system, potentially introducing other bugs. For this reason in multi-threaded hardware the analysis of the traces is performed offline [19]. Finally, due to the specific architecture of their thread-shared trace buffers, the methodology is not beneficial in case of homogeneous threads, like for example OpenMP for loops or `pthreads` executing the same function. In contrast, the methodology proposed here does not have these limitations, and it is specifically designed to handle homogeneous multi-threading gracefully.

Another trend in on-chip debugging is to synthesize ANSI-C assertions, generating checker circuits. In this field there are works that focus on the architecture of the assertion checkers [12, 23, 41], as well as on how to physically place them on FPGA without affecting latencies [26]. These methodologies are effective, but they can only check malfunctions foreseen by the developers, because assertions must be manually inserted in the original `C` specification. This fails to spot bugs that are not guarded by assertions, whereas the methodology proposed in

this article provides visibility on such bugs and automated identification of their root cause. In addition, none of the discussed works on assertion-based verification for HLS mention support for coarse-grained parallel programming paradigm.

Another big group of methodologies similar to Discrepancy Analysis provides automated bug detection comparing the execution of the circuits generated with HLS and of the software obtained from their original high-level specifications, with simulation or directly on-chip. Campbell et al. [8] focus on Application Specific Integrated Circuit. 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 (which they call hardware signatures) from the circuit. The golden reference and the hardware signature are then compared at the end of the execution and bugs are automatically detected. Campbell et al. [7] use the same methodology to FPGA, but differently from Reference [8] they rely on simulation for the generation of the hardware signatures. Yang et al. [50, 51], instead, actually use the golden reference obtained from the IR to generate the RTL instrumentations, but the whole debugging flow still relies on simulation. However, differently from References [7] and [14], the comparison between hardware behavior and software behavior is not performed at the end of the execution, but is executed concurrently by the RTL instrumentation during simulation; Carrion Schafer in Reference [42] does the same. On the contrary, Calagar et al. [6] analyze the discrepancies online, during the executions of hardware and software. Their proposed work exploits both simulation and on-chip debugging. They do not generate the golden reference, but instead they use a conventional debugger to observe the software on the fly, Application Programming Interfaces of the simulator to the simulator APIs to analyze the simulated RTL, and Altera SignalTap for in-circuit debugging. Their work, however, does not support most of the compiler optimizations performed during the HLS, and the use of SignalTap causes a high memory usage for the trace buffers, as reported also in Reference [33]. Iskander et al. [27] propose an 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 Low-level Debug, instead, uses partial reconfigurability to provide observability, insert breakpoints, and provide a software-like debugging experience. The main big limitation of all these approaches for automated bug detection, as well as of our previous work [14], is that they do not consider multi-threaded programs and they cannot cope with the large variety of hardware/software thread mappings to properly compare the executions.

As for methodologies for debugging hardware generated from multi-threaded programs, one of the few contributions (besides the work of Goeders et al. mentioned above [19]) is a work of Verma et al. [47] targeting OpenCL for FPGAs. The authors describe open-source debug components, modeled both in the OpenCL language and in Verilog Hardware Description Language (HDL), that can be used for manual inspection of OpenCL kernels running on FPGA. The work focuses on the architecture and on providing these components as a key enabling technique for increasing visibility on signals during execution. They do not discuss if and how the information collected with their method can be analyzed automatically for bug detection and source-level backtracking.

Automated bug detection has attracted much interest, as demonstrated by the variety of different flavors described above. Unfortunately, these works do not consider the problem of debugging hardware generated from multi-threaded parallel programs. This scenario introduces a number of challenges when trying to put into relationship the execution of the multi-threaded software with the execution of the parallel hardware implementation generated with HLS. The reason is that the number of threads and the actual mapping between task and threads can be different in software and in hardware. Depending on the configuration and the optimizations implemented by the HLS

tool, two tasks that are executed by the same thread in the original software could be mapped onto two physically distinct instances of the HW component. However, the software could launch a large number of threads, while the design generated from HLS could throttle physical parallelism to contain area consumption on FPGA.

All these problems are not taken into account by the approaches discussed above and are exacerbated with irregular applications, where the hardware/software task mapping patterns are much harder to predict. In References [19] and [47], the task of unraveling this complexity is delegated to users that have to figure out the particular thread mapping decided by HLS. Things are complicated by the fact that for certain programming models the thread mapping in software is decided by the language runtime and is not deterministic. The approach described in this work specifically targets thread parallel programming models, trying to tackle these problems. Unlike References [19] and [47], it does not focus on the architecture of the debugging components but it describes an efficient methodology for automated bug detection and source-level backtracking, specifically designed to handle circuits generated with HLS from parallel programs.

The fact that there have been so many recent contributions in the field of bug detection on FPGA shows that it is an interesting open problem. The approach presented here advances the state-of-the-art of bug detection on FPGA, *specifically for parallel programming paradigms* whose precise debugging have long been neglected on FPGA, as testified by the fact that very few of the mentioned techniques tackle this problem, and the few who do also degrade heavily when debugging OpenMP programs.

In particular, the novelty of the approach presented here is that it is able to provide *at the same time* all the following features: (1) *trace-based fully automated bug detection* without user interaction; (2) *independence of the technique used for collection of the traces*, which can be collected on-chip or with simulation as shown in References [14, 16]; (3) *full support for parallel programming paradigms on FPGAs* that do not degrade even in presence of complicated dynamic mappings between software threads and replicated hardware components. This is the first approach that provides all these features on FPGA without being tied to a specific implementation of hardware/software thread mapping or task-scheduling mechanism.

8 CASE STUDIES AND RESULTS

This section describes a proof-of-concept implementation of the described methodology and evaluates it on OpenMP benchmarks to show advantages and limitations of the approach.

8.1 Proof-of-concept Implementation

The implementation has been developed as part of the PandA open source framework for HW/SW co-design, developed at Politecnico di Milano. The framework includes an HLS compiler, BAMBU [35], which can generate Verilog or VHDL code starting from `C` specifications with support for OpenMP pragmas, and that can automatically instrument the code and detect bugs using the described approach. For the purpose of this work, the BAMBU debug flow depicted in Figure 9 has been extended to extract and handle the additional information necessary for the comparison of the traces coming from multi-threaded programs and the resulting hardware designs. The portion involving the software traces is in blue on the left. The `C` code with instrumentations and restructured in SSA is compiled and executed for the generation of the Software Traces. GCC-4.9 is used for compilation. The portion involving hardware traces is on the right in orange. Then the RTL generated by BAMBU is simulated with cycle accuracy for the generation of the Hardware Traces. The results shown in the following have been generated using ModelSim SE-64 10.5 from Mentor Graphics for simulation, but there is nothing specific to this simulator in the process. The green part in the middle shows information flowing from HLS to the Discrepancy Analysis bug detection

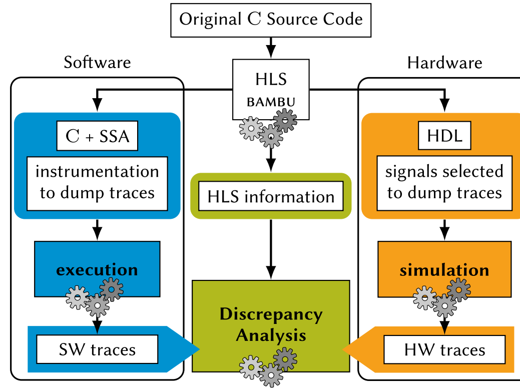


Fig. 9. Outline of the Discrepancy Analysis debug flow.

step. This last step collects the traces and analyzes them without user interaction. If a mismatch is detected, information regarding the failing operations and the involved threads in hardware and in software is provided to the user. This flow has been tested to locate bugs in circuits generated by BAMBU for a set of OpenMP benchmarks.

8.2 Benchmarks

We evaluated our methodology on a set of benchmarks composed of seven `C` programs that use parallel programming directives.

This specific set of benchmarks has been selected for two main reasons. *First*, because it has already been used to evaluate HLS of OpenMP programs in other research work targeted to FPGAs (e.g., Choi et al. [11]). *Second*, because it is representative of the real current use of the OpenMP programming model on FPGA.

As mentioned in Section 2.2, FPGAs are very resource-constrained platforms compared to the typical HPC machines where parallel programming languages are mostly used, which do not have severe limits on memory nor on power consumption. In contrast to the typical HPC use, OpenMP on FPGAs is mostly used for offloading to hardware small kernels, which can be easily parallelized to exploit physically parallel computation on FPGA. This can be very beneficial for performance and for power consumption.

The benchmarks are the following:

- *Black-Scholes* (bs): fixed point computation for option pricing with Monte Carlo approach.
- *Division* (div): divides a set of integers in an array by another set of integers.
- *Floating Point Sine Function* (dfsin): adopted from the CHStone benchmark suite [24], it implements a double-precision floating-point sine function using 64-bit integers.
- *Hash* (hash): uses four different integer hashing algorithms to hash a set of numbers and compares the number of collisions caused by the four different hashes.
- *Line of Sight* (los): uses the Bresenham's line algorithm to determine whether each pixel in a two-dimensional grid is visible from the source.
- *Mandelbrot* (mb): an iterative mathematical benchmark that generates a fractal image.
- *MCML* (mcml): light propagation from a point source in an infinite isotropic medium.

Some of the benchmarks contained a mix of OpenMP and `pthreads` directives in Reference [11], but we adapted them to only use OpenMP. We used BAMBU with its default parameters to generate Verilog designs for all the benchmarks, adding the `-fopenmp` flag to enable OpenMP. In

particular, the generated architecture dynamically dispatches the tasks onto the various copies of the accelerators. In addition to this, we used dedicated components to simulate accesses with variable latency during the operations of the algorithms to inject more irregularity in the computations. The default target device is a Xilinx Zynq-7000 xc7z020-1clg484, with a frequency of 100 MHz. We ran the integrated co-simulation flow to ensure that the generated hardware was working properly. Then, we manually injected different kinds of bugs (see Section 8.3) to see if they could be detected. We also ran the bug detection on the unmodified designs to check for false positives and to measure its overhead.

8.3 Bug Detection

We tested the method manually inserting three different kinds of bugs. The *first* class is composed by bugs located in a *single hardware thread*. For these bugs, the capabilities of the bug detection are the same as for the regular Discrepancy Analysis for single-threaded programs. This means that it finds bugs affecting each single thread with the same accuracy of the single-threaded version, even if the application is irregular and the generated design is multi-threaded and independently of the HW/SW task mapping. This holds both for control flow bugs and for faults involving single operations. For custom data types, the approach can use special comparison functions, for instance, considering Unit in Last Place for floating points or considering the HW/SW address mapping for pointers (see Reference [15]). The approach can also isolate bugs in libraries of external components used as elementary operators in HLS.

The *second* class of bugs involves *communication between threads* via shared memory. The extended Discrepancy Analysis detects situations where thread accelerators' reads of writes wrong values to or from memory, as well as when thread accelerators access memory at wrong locations. One example is when an accelerator accesses a portion of the shared memory that is reserved for another thread. Another example is when an accelerator accesses a global data structure instead of its own thread-private copy. Our proof-of-concept was always able to find these bugs when injected. Other reported communication bugs are caused by thread synchronization and non-deterministic locking order. This last class of bugs is actually a false positive and is discussed in Section 8.5.

The *third* class of bugs was caused by *missed or multiple executions of tasks*. The Discrepancy Analysis detects if a given task is executed a different number of times in hardware and in software. This may happen due to bugs in the logic of the component that decides which task has to be executed on a given physical copy of the thread accelerator. The detection works if a given task is dispatched multiple times on different copies of the thread accelerator, as well as on the same copy. It is also able to detect if a given task executed in software is never executed in hardware.

It is worth to notice that with multi-threaded Discrepancy Analysis the strong guarantee that the detected bug is the first is lost. One reason is that, in absence of a serial execution and with possibly different thread models in hardware and in software, it is possible to give different definitions of "first." The other reason is that with the trace mangling described in Section 6 the absolute global timeline of the simulation is scattered through the filtered traces. Timing information is preserved, but every filtered trace maintains only part of it. The result is that at first it is only possible to identify the first mismatch for each task. Then the global timestamps of each mismatch for each task have to be compared to decide which happened first in hardware.

8.4 Performance and Other Advantages

The performance of multi-threaded Discrepancy Analysis was evaluated measuring its execution time when the generated design was bug-free. This enabled to measure the real execution time of the algorithm, because in presence of bugs the comparison of each trace is skipped after the first

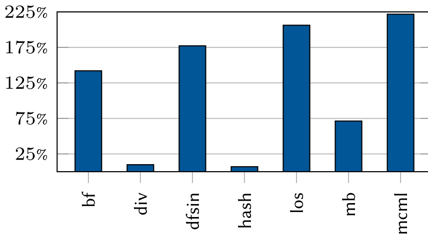


Fig. 10. Time overhead of the Discrepancy Analysis, compared to the simulation time.

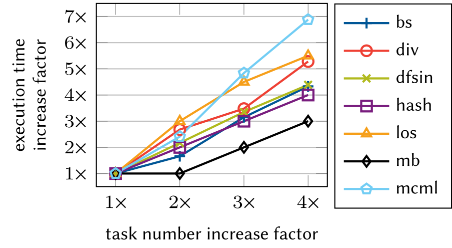


Fig. 11. Correlation between length of execution traces and execution time of the Discrepancy Analysis.

mismatch. Without bugs, the debugger is forced to analyze all the traces to the end. The results are obtained from the designs generated by BAMBU on the set of benchmarks described in Section 8.2. The simulation was executed with ModelSim SE-64 10.5 from Mentor Graphics. To measure the overhead of this debugging approach, we compared the execution time of the bug detection to the simulation time. The results are reported in Figure 10. The simulation times for the evaluated benchmarks were always in the order of a few tens of minutes, so the overhead of the bug detection was acceptable. However, it is evident that there is a large variance depending on the benchmark. On div and hash the overhead is only about 10%. For bf, dfsin, los, and mcml, instead, it grows above 100% up to about 225%. There are various reasons for these differences, but they are to be attributed to two main causes.

The first is that BAMBU generates very different architectures for the parallel constructs due to optimizations. In particular, there is a very different degree of resource sharing. Accelerators with multiple duplicated components generate a larger number of hardware traces, while for accelerators with heavy sharing this number is limited. With a larger number of traces the work that the debugger has to perform is much bigger, hence the great increase in the overhead.

The second reason is related to the memory architecture. For benchmarks with larger memories and a higher number of shared variables the number of memory accesses and pointer operations is also higher. This triggers the address Discrepancy Analysis algorithm implemented by BAMBU and described in Reference [15]. This algorithm is more complicated, because it keeps track of context-dependent memory locations of stack-allocated variables in software to build tables that are queried by the Discrepancy Analysis to resolve matches and mismatches on pointer operations. This is the second cause of the large variance in overhead.

However, a large overhead is not necessarily to be interpreted as a negative outcome. Given that large overheads are measured on complicated designs, this overhead actually measures the amount of work that a designer should perform manually to find bugs in such designs. The automated bug detection is clearly an advantage in these cases, because it avoids user interaction and it is suitable for continuous integration and regression testing. It is also interesting to see how the Discrepancy Analysis scales in case of long runs. To measure it, we executed it on multiple runs of the same designs, varying the workload of the multi-threaded part. Figure 11 reports data on how the execution time increases with the increase of the multi-threaded workload. In general, the execution time grows roughly linearly with the workload, with slightly different slopes depending on the design.

Another advantage of the Discrepancy Analysis is that it automatically selects the signals necessary to generate the Hardware Traces. Without it, developers have to dump the complete traces of all the signals in the design and inspect them manually. This often leads to waveform files of unmanageable size. With Discrepancy Analysis only the necessary signals are actually

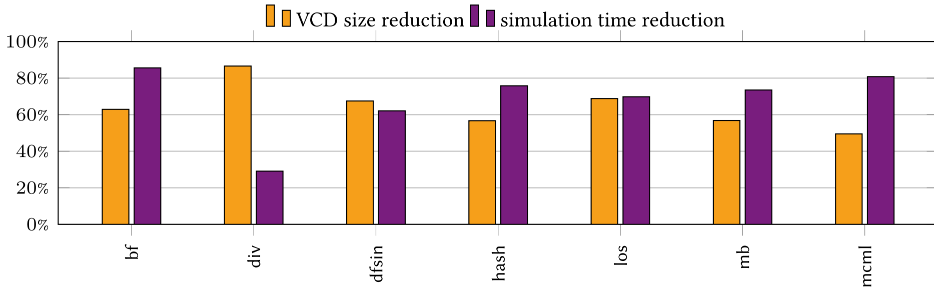


Fig. 12. Reduction of VCD size and simulation time when Discrepancy Analysis is enabled.

printed, decreasing the impact of the I/O operations on the simulation. Figure 12 reports two datasets: (1) the reduction of the size of the VCD files with Discrepancy Analysis; (2) the reduction of simulation time. Both the data come from simulations with ModelSim SE-64 10.5 on the designs generated by BAMBU for the evaluated benchmarks. As we can see, the reduction of VCD size is always at least 50%, with peaks of more than 80%. In some cases this makes the difference between GBytes and MBytes and allows to analyze executions that are otherwise too long. The reduction of VCD size is reflected by the reduction of simulation times. The correlation between the two values is not always evident, such as for div and dfsin. The reason is that the simulator is able to optimize the design before simulation. Excluding the time spent in I/O for the creation of VCD, the simulation has some other fixed cost for initialization, static optimization, and other similar operation. These fixed costs are more significant on smaller benchmarks and cannot be avoided with signal selection. In fact, the benchmarks whose simulation is sped up more are the biggest. For those cases the fixed costs are less significant, and the advantages of signal selection are heavier. This is good for scalability, because the speedups are greater for bigger designs.

8.5 False Positives and Other Limitations

The major limitation of the approach is that the algorithm for debugging multi-threaded code described in Section 6 assumes that tasks assigned to each physical or logical thread are uniquely identified by a possibly dynamic task identifier. This is reasonable with homogeneous parallelism such as with OpenMP for loops, OpenCL NDRanges, and CUDA warps, but it is not always true in high-level multi-threaded parallel programs. Imagine a scenario with a single producer and multiple consumers, where the producer enqueues non-unique data to be processed by the consumers. The time necessary to process each element is not known in advance and can vary. In software as in hardware, thread IDs are not enough to know which thread is actually doing what, even with runtime data. The reason is that, depending on the latencies, each element in the queue could be processed by any thread, both in hardware and in software. To know which hardware and software threads are processing a specific element in the queue, one should not rely on thread IDs. Task IDs are not even present, so the only way to know it is to look at the actual data being processed. If the data in the queue are not unique, this is not possible with the approach described in Section 6.

The same holds in presence of synchronization directives, such as locks and critical sections with non-deterministic outcomes, such as a shared counter incremented atomically by every thread. In this case, the order of the increments is irrelevant for the correctness, as long as all the increments are actually atomic and the final value of the counter matches. This practically means that the results of the increments in hardware and software are not required to match for correctness, but Discrepancy Analysis has no way to know it. A simple workaround with OpenMP is to use

local counters with the `reduction` clause as in Figure 1, or with user-defined reduction. For more complex use cases this may not be entirely possible and is definitely worth further investigation.

These limitations practically limit the applicability of the proposed methodology to large OpenMP programs with rich semantics. However, as mentioned in Section 2.2, the real-world use in HLS and on FPGA of parallel programming models practically boils down to well-isolated kernels, which do not suffer these limitations. The current state-of-the-art of HLS for parallel programming directives still does not support richer behaviors and still does not cover the full expressive power of OpenMP semantics. Until HLS tools actually support synthesizing programs with richer semantics, it is premature to try to extend the proposed approach to support them, because no assumption on the actual hardware implementations can be done yet.

9 CONCLUSION AND FUTURE WORK

As HLS gains traction and FPGAs become interesting in massively parallel application, a growing number of HLS tools have started to provide support for high-level parallel programming languages. The work presented here describes a methodology for automated bug detection in hardware generated with HLS from parallel programs. The proposed approach allows to perform fast and effective Discrepancy Analysis between hardware and software with operation granularity and independently of the number of threads. The effectiveness of the methodology has been tested on parallel OpenMP benchmarks, but the discussion of the approach has been taken at a higher level to encompass a large class of high-level languages and threading directives. The proof-of-concept developed to evaluate the approach has proved to be valuable, finding several different classes of bugs involving errors in single threads, communication between threads, and wrong dispatch of thread iterations. The implementation proposed here is based on simulation to demonstrate the approach, but the technique is also usable with traces collected on-chip. However, debugging multi-threaded code is hard and there is room for improvement, avoiding false positives or extending support for programming models. These directions will be explored in our future research.

REFERENCES

[1] IEEE. 2016. Standard for information technology—Portable operating system interface (POSIX) base specifications, Issue 7. *IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)* (Sept. 2016), 1–3957.

[2] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp. 2008. Achieving programming model abstractions for reconfigurable computing. *IEEE Trans. Very Large Scale Integ. (VLSI) Syst.* 16, 1 (Jan. 2008), 34–44.

[3] The OpenMP Architecture Review Board. 2015. OpenMP Application Programming Interface—Version 4.5. Retrieved from http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[4] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. 1998. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.* 28, 8 (July 1998), 859–881.

[5] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, and D. Jimenez-Gonzalez. 2009. OpenMP extensions for FPGA accelerators. In *Proceedings of the International Symposium on Systems, Architectures, Modeling, and Simulation.* 17–24.

[6] N. Calagar, S. D. Brown, and J. H. Anderson. 2014. Source-level debugging for FPGA high-level synthesis. In *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL'14).* 1–8.

[7] K. Campbell, L. He, L. Yang, S. Gurumani, K. Rupnow, and D. 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'16).* ACM, New York, NY.

[8] K. Campbell, D. Lin, S. Mitra, and D. 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 (DAC'15).* ACM, New York, NY.

[9] V. G. Castellana and F. Ferrandi. 2013. An automated flow for the high level synthesis of coarse grained parallel applications. In *Proceedings of the International Conference on Field-programmable Technology (FPT'13).* 294–301.

[10] Vito Giovanni Castellana, Marco Minutoli, Antonino Tumeo, Marco Lattuada, Pietro Fezzardi, and Fabrizio Ferrandi. 2019. Software defined architectures for data analytics. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASPDAC'19)*. 711–718. DOI: https://doi.org/10.1145/3287624.3288754

[11] J. Choi, S. Brown, and J. Anderson. 2013. From software threads to parallel hardware in high-level synthesis for FPGAs. In *Proceedings of the International Conference on Field-programmable Technology (FPT'13)*. 270–277.

[12] J. Curreri, G. Stitt, and A. D. George. 2010. High-level synthesis techniques for in-circuit assertion-based verification. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW'10)*. 1–8.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.* 13, 4 (Oct. 1991), 451–490.

[14] P. Fezzardi, M. Castellana, and F. Ferrandi. 2015. Trace-based automated logical debugging for high-level synthesis generated circuits. In *Proceedings of the 33rd IEEE International Conference on Computer Design (ICCD'15)*. 251–258.

[15] P. Fezzardi and F. Ferrandi. 2016. Automated bug detection for pointers and memory accesses in high-level synthesis compilers. In *Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL'16)*. 1–9.

[16] P. Fezzardi, M. Lattuada, and F. Ferrandi. 2017. Using efficient path profiling to optimize memory consumption of on-chip debugging for high-level synthesis. *ACM Trans. Embed. Comput. Syst. (Special Issue on ESWEEK'17)* 1–19. Retrieved from https://re.public.polimi.it/retrieve/handle/11311/1030731/222692/EPPDiscrepancyAnalysis.pdf.

[17] Intel FPGA. 2017. Intel FPGA SDK for OpenCL—Programming Guide. Retrieved from https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf.

[18] J. Goeders and S. J. E. Wilton. 2015. Using dynamic signal-tracing to debug compiler-optimized HLS circuits on FPGAs. In *Proceedings of the IEEE 23rd Annual International Symposium on Field-programmable Custom Computing Machines (FCCM'15)*. 127–134.

[19] J. Goeders and S. J. E. Wilton. 2015. Using round-robin tracepoints to debug multithreaded HLS circuits on FPGAs. In *Proceedings of the International Conference on Field Programmable Technology (FPT'15)*. 40–47.

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

[21] Khronos OpenCL Working Group. 2017. The OpenCL Specification—Version 2.2. Retrieved from https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf.

[22] R. J. Halstead and W. Najjar. 2013. Compiled multithreaded data paths on FPGAs for dynamic workloads. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'13)*. IEEE Press, Piscataway, NJ. Retrieved from http://dl.acm.org/citation.cfm?id=2555729.2555732.

[23] M. B. Hammouda, P. Coussy, and L. Lagadec. 2017. A unified design flow to automatically generate on-chip monitors during high-level synthesis of hardware accelerators. *IEEE Trans. Comput.-aided Des. Integ. Circ. Syst.* 36, 3 (Mar. 2017), 384–397.

[24] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. 2008. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'08)*. 1192–1195.

[25] M. Hosseinabady and J. L. Nunez-Yanez. 2015. Optimised OpenCL workgroup synthesis for hybrid ARM-FPGA devices. In *Proceedings of the 25th International Conference on Field Programmable Logic and Applications (FPL'15)*. 1–6.

[26] Eddie Hung, Tim Todman, and Wayne Luk. 2017. Transparent in-circuit assertions for FPGAs. *IEEE Trans. CAD Integ. Circ. Syst.* 36, 7 (2017), 1193–1202.

[27] Y. Iskander, C. Patterson, and S. Craven. 2014. High-level abstractions and modular debugging for FPGA design validation. *ACM Trans. Reconfig. Technol. Syst.* 7, 1 (Feb. 2014).

[28] Al-Shahna Jamal, Jeffrey Goeders, and Steven J. E. Wilton. 2018. Architecture exploration for HLS-oriented FPGA debug overlays. In *Proceedings of the ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA'18)*. ACM, New York, NY, 209–218. DOI: https://doi.org/10.1145/3174243.3174254

[29] J. Korinth, D. de la Chevallerie, and A. Koch. 2015. An open-source tool flow for the composition of reconfigurable hardware thread pool architectures. In *Proceedings of the IEEE 23rd Annual International Symposium on Field-programmable Custom Computing Machines*. 195–198.

[30] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. 75–86.

[31] S. Ma, M. Huang, and D. Andrews. 2012. Developing application-specific multiprocessor platforms on FPGAs. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*. 1–6.

[32] M. Minutoli, V. G. Castellana, A. Tumeo, M. Lattuada, and F. Ferrandi. 2016. Enabling the high level synthesis of data analytics accelerators. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'16)*. 1–3.

[33] J. S. Monson and B. L. Hutchings. 2014. New approaches for in-system debug of behaviorally-synthesized FPGA circuits. In *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL'14).* 1–6.

[34] J. S. Monson and Brad L. Hutchings. 2015. Using source-level transformations to improve high-level synthesis debug and validation on FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA'15).* ACM, New York, NY, 5–8.

[35] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. 2016. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans. Comput.-aided Des. Integ. Circ. Syst.* PP, 99 (2016), 1–1.

[36] NEC. 2016. *CyberWorkbench: NEC's High Level Synthesis Solution.* Retrieved from http://www.nec.com/en/global/prod/cwb/pdf/CWB_Detailed_technical.pdf.

[37] T. Nguyen, Y. Cheny, K. Rupnow, S. Gurumani, and D. Chen. 2016. SoC, NoC and hierarchical bus implementations of applications on FPGAs using the FCUDA flow. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'16).* 661–666.

[38] NVIDIA. 2017. CUDA Parallel Programming and Computing Platform. Retrieved from http://www.nvidia.com/object/cuda_home_new.html.

[39] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos. 2011. Synthesis of platform architectures from OpenCL programs. In *Proceedings of the IEEE 19th Annual International Symposium on Field-programmable Custom Computing Machines.* 186–193.

[40] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA'14).*

[41] A. Ribon, B. Le Gal, C. Jégo, and D. Dallet. 2011. Assertion support in high-level synthesis design flow. In *Proceedings of the FDL Forum on Specification, Verification and Design Languages.* 1–8.

[42] B. Carrion Schafer. 2016. Source code error detection in high-level synthesis functional verification. *IEEE Trans. Very Large Scale Integ. (VLSI) Syst.* 24, 1 (Jan. 2016), 301–312.

[43] R. M. Stallman and GCC Developer Community. 2009. *Using the GNU Compiler Collection: A GNU Manual for GCC Version 4.3.3.* CreateSpace Independent Publishing Platform.

[44] A. Takach. 2016. High-level synthesis: Status, trends, and future directions. *IEEE Des. Test* 33, 3 (June 2016), 116–124.

[45] M. Tan, B. Liu, S. Dai, and Z. Zhang. 2014. Multithreaded pipeline synthesis for data-parallel kernels. In *Proceedings of the IEEE/ACM International Conference on Computer-aided Design (ICCAD'14).* 718–725.

[46] F. Vahid. 1997. Procedure cloning: A transformation for improved system-level functional partitioning. In *Proceedings of the European Design and Test Conference (EDTC'97).* 487–492.

[47] A. Verma, H. Zhou, S. Booth, R. King, J. Coole, A. Keep, J. Marshall, and W.-C. Feng. 2017. Developing dynamic profiling and debugging support in OpenCL for FPGAs. In *Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference (DAC'17).*

[48] Y. Wang, J. Yan, X. Zhou, L. Wang, W. Luk, C. Peng, and J. Tong. 2012. A partially reconfigurable architecture supporting hardware threads. In *Proceedings of the International Conference on Field-programmable Technology.* 269–276.

[49] Xilinx. 2017. The SDAccel Development Environment for OpenCL. Retrieved from https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[50] L. Yang, S. Gurumani, D. Chen, and K. Rupnow. 2016. AutoSLIDE: Automatic source-level instrumentation and debugging for HLS. In *Proceedings of the IEEE 24th Annual International Symposium on Field-programmable Custom Computing Machines (FCCM'16).* 127–130.

[51] L. Yang, M. Ikram, S. Gurumani, S. Fahmy, D. Chen, and K. Rupnow. 2015. JIT trace-based verification for high-level synthesis. In *Proceedings of the International Conference on Field Programmable Technology (FPT'15).* 228–231.

Received November 2018; revised November 2019; accepted June 2020