Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
Ph.D. Program in Information Technology

Discrepancy Analysis: a Methodology for Automated Bug Detection in Hardware Designs Generated with High-Level Synthesis

Doctoral Dissertation of:
Pietro Fezzardi

Supervisor:
Prof. Fabrizio Ferrandi

Tutor:
Prof. Cristiana Bolchini

The Chair of the Doctoral Program:
Prof. Andrea Bonarini

XXX cycle

Abstract

This thesis describes the definition, implementation, and evaluation of a methodology for automated bug detection, called *Discrepancy Analysis*, targeted at hardware designs generated with High-Level Synthesis. *Discrepancy Analysis* is based on a notion of equivalence between the execution of the hardware generated with High-Level Synthesis and the execution of the software obtained from the original high-level source code used to generate that hardware. Using this notion of equivalence, the thesis describes how to compare automatically the two executions, and how to detect and isolate the first mismatch if present. All these operations are executed without human interaction, relieving users from the time-consuming and error-prone tasks to select the necessary signals for debugging, analyzing the signal traces to identify the malfunction, and backtracking it to the original high-level source code.

The methodology is tightly integrated with the High-Level Synthesis process. As a consequence, it supports all compiler optimizations available during High-Level Synthesis. This coupling with the High-Level Synthesis tool also allows to automatically select in the generated designs the signals necessary for automated bug detection. Despite the tight coupling with the High-Level Synthesis tool, the discussion is kept as general as possible and only relies on common features that are present in all the known commercial and academic tools. The thesis also describes two extensions of *Discrepancy Analysis*: one to support automated bug detection in hardware generated with High-Level Synthesis of multithreaded code; one to support automated bug detection on pointers and memory accesses.

Two bug detection flows based on *Discrepancy Analysis* are presented. The first is based on simulation of the hardware at the Register Transfer Level and performs the automated bug detection process offline after execution. The second flow is for on-chip bug detection. The generated hardware is instrumented with dedicated checker components, that analyze the execution on the fly, halting the circuit if a mismatch occurs and notifying it to users. Both the debug flows have been implemented and tested with BAMBU, an open source research framework for High-Level Synthesis developed at Politecnico di Milano.

The results have been evaluated in terms of performance, coverage, and other advantages brought to the overall debugging experience, like the considerable reduction of the size of the waveforms files that can be achieved with a heuristic for automated signal selection. This evaluation showed *Discrepancy Analysis* to be fast, accurate, and effective in identifying several different classes of bugs, coming from the original high-level code, from external libraries of components, and even subtle bugs injected by the High-Level Synthesis tool itself. A thorough and extensive analysis of these classes of bugs has been carried on, both on the baseline version and on the presented extensions for multithreaded code, for pointers, and for on-chip debugging. The technique used to compress the execution traces for *On-Chip Discrepancy Analysis*, based on Efficient Path Profiling, also showed reductions of the memory consumption necessary for on-chip debugging up to 95% compared to previous state-of-the-art.

Part of the material contained in this work has been previously published in international peer-reviewed conferences and journals:

- P. Fezzardi, M. Castellana, and F. Ferrandi. Trace-based Automated Logical Debugging for High-Level Synthesis Generated Circuits. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 251–258, Oct 2015
- P. Fezzardi and F. Ferrandi. Automated Bug Detection for Pointers and Memory Accesses in High-Level Synthesis Compilers. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016
- P. Fezzardi, M. Lattuada, and F. Ferrandi. Using Efficient Path Profiling to Optimize Memory Consumption of On-Chip Debugging for High-Level Synthesis. *ACM Transactions on Embedded Computing Systems*, 16(5s):149:1–149:19, Sept. 2017

Contents

Abstract i

Contents iii

1 Introduction 1

I BACKGROUND 7

2 High-Level Synthesis (HLS) 11

2.1 Introduction to High-Level Synthesis 11
2.2 Memory Allocation and Hardware Synthesis of C Pointers 17
2.3 High-Level Synthesis of Multi-Threaded Programs 19

3 State of the Art of Debugging Methodologies for HLS 23

3.1 Concepts of Hardware Debugging 23
3.2 Challenges in Debugging Hardware Generated with HLS 25
3.3 Debugging Methodologies for High-Level Synthesis 27

II METHODOLOGY 33

4 Problem Statement and Goals 37

4.1 Description of the Problem 37
4.2 Fundamental Ideas of the Approach 37
4.3 Objectives, Goals, and Features 38
4.4 Detected Classes of Bugs 39

5 Equivalence Between Hardware and Software Execution 43

5.1 Control Flow Level 43
5.2 Operation Level 45
5.3 Hardware/Software Equivalence 47
5.4 A Generic Workflow for Discrepancy Analysis 48

6 Discrepancy Analysis: Two Different Flows 51

6.1 Reference Implementation 51
6.2 Simulation-Based Offline Discrepancy Analysis 52
6.3 On-Chip Online Discrepancy Analysis 53

7 Simulation-Based Offline Discrepancy Analysis 57

7.1 Generating and Collecting Execution Traces 57
7.2 Comparing Execution Traces with Finite State Automata 59
7.3 Debugging Circuits Generated from Multithreaded Programs 64

8 Debugging Pointers and Memory Accesses **69**
8.1 Address Space Translation Scheme . . . 69
8.2 Address Discrepancy Algorithm . . . 71
8.3 Refining Address Discrepancy Analysis . . . 72
9 On-Chip Online Discrepancy Analysis of Control Flow **75**
9.1 Motivation . . . 75
9.2 Efficient Path Profiling for Software . . . 76
9.3 Efficient Path Profiling for High-Level Synthesis . . . 79
9.4 Optimization of Memory Usage . . . 82
9.5 Architecture of the Control Flow Checkers . . . 83

III EXPERIMENTAL RESULTS **87**
10 Experimental Setup **91**
10.1 Integration with High-Level Synthesis . . . 91
10.2 Experiments and Benchmarks . . . 92
11 Detected Bugs **99**
11.1 Bugs Detected with Simulation . . . 99
11.2 Bugs Involving Addresses . . . 101
11.3 Bugs in Multi-Threaded Programs . . . 102
11.4 Bugs Detected On-Chip . . . 103
12 Simulation-Based Discrepancy Analysis **107**
12.1 Baseline . . . 107
12.2 Multi-Threaded . . . 115
12.3 Address Discrepancy Analysis . . . 118
13 On-Chip Discrepancy Analysis **125**
13.1 Memory Usage . . . 125
13.2 Overhead of the Tracing Logic . . . 127
13.3 Limitations of the Proposed Approach . . . 129
14 Conclusion and Future Research **131**

INDICES **I**
List of Figures **III**
List of Tables **V**
List of Algorithms **VII**
Bibliography **IX**

1 Introduction

Field Programmable Gate Arrays (FPGA) are steadily becoming more appealing in computing. They provide reconfigurability and flexibility similar to software solutions while guaranteeing low power consumption and massive physical parallelism close to what is possible with Application Specific Integrated Circuits (ASIC). These characteristics are very promising in the current general struggle to find new computational paradigms that can cope with the end of Moore's Law. For these reasons, FPGAs are not under investigation just for prototyping, but they are increasingly used in datacenters [70], High-Performance Computing and irregular parallel applications [55] [56].

One of the main obstacles to the mainstream adoptions of such devices is that the set of skills and competencies necessary to program them effectively is very broad. Skilled digital circuits designers are harder to find on the market than software engineers, and the development of a dedicated digital circuit to execute a given task usually requires considerably more time than a software implementation of the same functionality. FPGAs are traditionally programmed using so called Hardware Description Languages (HDL), that allow designers to describe specifically every component of the digital electronics that compose the system. For their nature, these languages are very tightly related to the underlying devices and electronics, forcing designers to focus at the same time on the low-level details of the electronics and on the high-level algorithmic level of their implementations. This close relationship with the underlying hardware also means that the same HDL design cannot be easily ported from an FPGA device to another, without significant adaptation to the new characteristics of the board. These three components – scarceness of skilled engineers, languages that place multiple heavy responsibilities on the shoulders of designers, and lack of portability of hardware designs – clearly represent a scalability problem that has to be overcome to enable FPGA computing to really go mainstream.

In recent years, a promising approach in this field that has received much attention is High-Level Synthesis (HLS). The main insight behind High-Level Synthesis is that the key to removing all the obstacles to FPGA design is to use a high-level software programming language, instead of HDL, as a starting point for hardware synthesis. With this approach software engineers could start programming FPGAs, ignoring the details of the underlying electronic designs to focus on algorithms, while at the same time creating designs that are easily portable on new devices. Today, many different academic and commercial tools are able to generate HDL design starting from a variety of programming languages: `C`, `C++`, Java, Python, Haskell, Erlang, and many others. However, the most common and well supported by FPGA vendors are `C` and `C++`. These languages are high-level

compared to HDL and allow designers to write portable code, while still giving programmers the capability to control low-level details that might be relevant in hardware. Moreover, `C` and `C++` are the basis of a large number of standard libraries and programming language extension for HPC and multithreading, like POSIX threads (`pthreads`), OpenMP, OpenCL, and CUDA. Commercial HLS tools and academic projects often provide support for High-Level Synthesis starting from these multi-threaded language extensions, providing programmers with familiar tools to easily exploit the massive physical parallelism available on FPGAs. These solutions are also seen as a promising trend to provide well-known programming paradigms for FPGAs in the cloud.

However, for a programming paradigm to succeed, there is more to be taken care of besides the programming language. One of the fundamental aspects that will determine the success or the failure of High-Level Synthesis in the long term is the ecosystem of development and verification tools that will be built around it. Indeed, in hardware design, testing and verification typically constitute a significant portion of the whole effort for a project. Another important factor is the support for the integration of multiple components in System-on-Chip (SoC) design. These components can be either generated with HLS, hand-written by designers, or provided as Intellectual Property (IP) blocks by third-parties. According to ITRS prediction [1] future SoC architectures will be characterized by heavy reuse (more than 90% by 2020) of IP blocks for reducing design cost and time-to-market. To increase productivity and tackle design complexity, designers will need to raise the abstraction level and use Electronic System Level (ESL) methodologies based on High-Level Synthesis to automatically generate and integrate the IP descriptions in a suitable HDL design [51]. This will result in a proliferation of IP vendors specialized in the optimization of specific functionalities, while system designers will focus on the integration of the different components, posing new threats for the design and verification of complex architectures. At the same time HLS compilers are growing in complexity, adding more optimizations passes to generate more efficient accelerators in term of frequency, area, or power consumption. This complexity is hidden to users and managed by the tools, but it can become a real burden during testing, debugging and verification. Given that in SoC design up to more than 50% of the overall time can be spent on verification [3], the risk is that the speedup HLS gives to development could be negatively compensated by a slowdown in testing and debugging. HDL generated by HLS is not intended to be human-friendly, because that is not the purpose of HLS. This may become a problem if the HLS users are software engineers with little previous exposure to HDL and to hardware design.

In order to avoid this risk, or even to improve the testing and debugging experience as much as the development phase has been improved, it is critical for High-Level Synthesis tools to integrate techniques and workflows to also manage verification. In particular, the tools need to keep track of the additional complexity introduced and managed by HLS during the design stage, in order to be able to reason about it later during verification, helping users to unravel the details of what went wrong in case of bugs. In this way, it is possible to extend the support that these tools give to the designers beyond implementation phase, up to the testing, debugging and verification steps.

This thesis focuses on one of the several facets of hardware testing and verifi-

cation: bug detection and isolation. Some of the main challenges that need to be faced in this area are the following:

1. manage complexity on behalf of users;
2. help and guide users in bug detection and isolation;
3. identify relevant signals in hardware and backtrack bugs to high-level code;
4. handle compiler optimizations and bugs introduced by HLS optimizations;
5. handle different hardware/software memory architectures and mappings;
6. handle HLS of multithreaded code;
7. handle integration of external components.

Points 1, 2, 3, and 4 represent the main obstacles users initially face when starting to debug circuits generated with High-Level Synthesis. The designs generated with HLS are often cryptic and hard to interpret especially if the compiler performed optimizations. Hence, users need help to understand how the original source code was mapped to hardware, what are the relevant signals, and what decisions were taken by the HLS tool during the process. Points 4, 5, and 6 involve some of the most delicate tasks of High-Level Synthesis compilers. It is somehow natural to expect from a debugging environment for HLS to be able to handle these scenarios and manage them from the users also during bug detection. After all, it is only the tool that knows exactly which kind of optimizations, memory mapping and thread mapping were encoded in the generated hardware. Finally, points 6 and 7 are necessary because they represent the direction where the industry is headed in the field of HLS. For these reasons, it is necessary to design bug detection methodologies in order to provide a good development experience for the upcoming design scenarios.

The subject of this thesis is a methodology for automated bug detection and identification in digital circuit designs generated with High-Level Synthesis for FPGAs. The technique is called *Discrepancy Analysis* and it is founded upon a definition of equivalence between the execution of the hardware generated with HLS and the software obtained from the same original source code. When the same input is provided to these two artifacts, the behavior of the hardware must be equivalent to the original specification. *Discrepancy Analysis* performs this comparison automatically, without user interaction, and at two levels of abstraction: Control Flow, and Data Flow. These two levels represent the two fundamental ways hardware and software execution can differ. The fine-grained checks can inspect every single operation, providing visibility onto temporary variables inserted by the compiler for optimizations. The automated bug detection algorithm is able to detect the first mismatch between hardware and software execution, providing to the users all the information available to HLS on the involved signals and high-level variables. This masks the complexity while giving back to users only the useful information improving the debugging experience. The approach is designed to use HLS information to support all the available compiler optimizations, different memory architectures, as well as multi-threaded code. The thesis defines the methodology from the ground up, starting with the definition of equivalence between hardware and software executions, and extending it

to support multi-threaded code and C pointers. It also describes the implementation of two separate flows for automated bug detection based on *Discrepancy Analysis*: *Simulation-Based Offline Discrepancy Analysis* and *On-Chip Online Discrepancy Analysis*. These two flows show the adaptability of the approach to different scenarios and use cases. Both the implementations have been extensively tested, showing the effectiveness of the techniques in finding and isolating different kinds of bugs: faults coming from user code, bugs in third-party IPs, errors due to a wrong implementation of compiler optimizations. This showed that the approach can be very useful both for developers of High-Level Synthesis tools and for designers using HLS for their work.

The rest of the thesis is divided into three parts: Part I provides the necessary background concepts and a review of the state-of-the-art; Part II completely introduces *Discrepancy Analysis*, the methodology proposed in the thesis; Part III analyzes and discusses the results obtained evaluating *Discrepancy Analysis* in a variety of scenarios, ranging from single-threaded to parallel code, and from simulation-based to on-chip bug detection.

Part I is divided into two chapters. Chapter 2 introduces the fundamental concepts behind High-Level Synthesis and some of its aspects that are more interesting for this work: hardware synthesis of C pointers, and High-Level Synthesis of multi-threaded programs. Chapter 3, describes the main challenges of debugging hardware generated with High-Level Synthesis, and provides an overview of the state-of-the-art techniques that try to solve this problem.

Part II defines from the ground up the methodology that is the core of the thesis: *Discrepancy Analysis*. Chapter 4 describes the problems that *Discrepancy Analysis* is set to solve, sketching the main ideas, goals, and features. Chapter 5 introduces the concept of equivalence between hardware and software executions on a given input, which is at the foundation of *Discrepancy Analysis*. Chapter 6 describes two different flows for automated bug detection with *Discrepancy Analysis*: one based on simulation for offline bug detection, the other for online bug detection directly on-chip. Chapter 7 describes the first flow, based on simulation, describing how to collect and compare execution traces from hardware and software. It also extends the approach to enable automated bug detection on hardware generated with High-Level Synthesis from multi-threaded programs. Chapter 8 further extends the flow, explaining how to resolve the intrinsic difference between hardware and software memory architectures and address spaces, to handle bugs involving pointers and memory accesses. Finally, Chapter 9 describes how to use and adapt a profiling technique called *Efficient Path Profiling* to bring *Discrepancy Analysis* on-chip for online bug detection. The chapter also describes the architecture of the components that are embedded in the generated designs to perform the checks.

Part III provides data about the experimental evaluation of *Discrepancy Analysis* in its different flavors. Chapter 10 describes how the methodology proposed in *Part II* has been implemented and integrated into a Free and Open Source High-Level Synthesis, and the set of benchmarks and tools that have been used to evaluate it. Chapter 11 lists the different classes of bugs that the proposed methodology can catch in different scenarios, providing some practical examples. Chapter 12 reports and discusses results obtained with the *Offline Simulation-Based Discrepancy Analysis*, both on the baseline version, on the extension to multi-threaded

programs, and on the extension to pointers and memory accesses. Chapter 13, instead, describes the evaluation of the *Online On-Chip Discrepancy Analysis*. Finally, Chapter 14 summarizes the results, highlighting the main advantages of the approach and outlining possible new directions of investigation.

PART I

BACKGROUND

This part provides the necessary introduction to the elementary concepts at the foundations of the work presented in the rest of the thesis.

Chapter 2 introduces High-Level Synthesis, initially describing the process as a whole, then focusing on some of its aspects that are relevant for this work.

Chapter 3 discusses the state-of-the-art in the field of debugging methodologies for High-Level Synthesis, pointing out what are the main challenges that the High-Level Synthesis poses in addition to traditional hardware debugging.

2

High-Level Synthesis (HLS)

High-Level Synthesis (HLS), sometimes also referred as *behavioral synthesis*, is an automated design process for digital electronics that starts from an algorithmic description expressed in a high-level programming language. The main topic of this thesis is a methodology for automated bug detection for circuits generated with HLS. This chapter describes some of the concepts behind High-Level Synthesis that will be used throughout the rest of the work. Section 2.1 outlines the fundamental steps and operations involved in the HLS process, introducing concepts that will be used in Chapter 5 to define the equivalence between hardware and software execution. Section 2.2 describes the hardware synthesis of `C` pointers, that will be necessary to extend the proposed methodology to pointers and addresses, like explained in Chapter 8. Section 2.3 discusses methodologies for the generation of parallel hardware designs starting from multi-threaded specifications, that will be used to understand how the Discrepancy Analysis described here can be adapted to debug circuits generated from multi-threaded programs, as explained in Section 7.3.

2.1 Introduction to High-Level Synthesis

High-Level Synthesis is a design methodology for digital circuits that automatically generates micro-architectures for hardware components starting from algorithmic specifications expressed in high-level software programming languages. In a typical HLS flow, the input source language is a restricted dialect of a programming language such as `C` [44] or `C++` [45], used to capture the behavioral description of the design. The output of the flow is a circuit design expressed in a

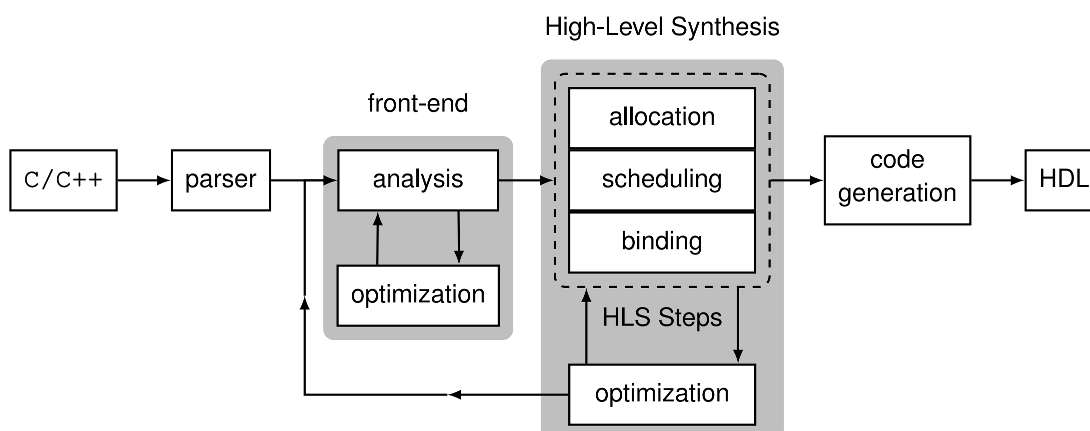


Figure 2.1: Structure of a typical High-Level Synthesis flow.

Hardware Description Language (HDL), such as VHDL [39] or Verilog [38], that describes the Register-Transfer Level (RTL) structure of the resulting hardware.

A High-Level Synthesis tool is essentially a compiler. The general structure of an HLS flow is depicted in Figure 2.1. The process unfolds from left to right. The input is a program specification in `C` or `C++`. The source code is read by a parser that usually generates an Intermediate Representation (IR), such as an Abstract Syntax Tree (AST). In general, different kinds of IRs are generated and used in different steps of the compilation process and HLS. The remainder of this section describes these steps in detail along with IRs they use, in particular those that are more relevant for the methodology described in the rest of the thesis.

The AST is passed to the first main stage of the compilation process: the *front-end*. Notice that what is called *front-end* here is actually what is usually called middle-end in traditional compilers and it includes several different analyses and optimizations on the IR. The name *front-end* is used to distinguish it from the core of the High-Level Synthesis process.

The results of the analyses and optimizations performed by the *front-end* are then passed to the HLS engine, which performs three main operations: *allocation*, *scheduling*, and *binding*. *Allocation* selects functional units to be used to perform the elementary operations contained into the IR. *Scheduling* maps each operation of the IR to the clock cycle when it will be executed. *Binding* assigns operations to specific instances of the functional units used to implement them, as selected from *allocation*. These three steps are actually tightly interdependent, and the overall HLS process is actually composed of other minor tasks. Moreover, there are some optimizations that can be triggered only after that the results for *allocation*, *scheduling* and *binding* have been calculated, or that are actually part of the process. As a consequence, it may be necessary to restart the *front-end* to propagate some of the new information across the IR, possibly enabling further improvements to the IR and to the results of HLS.

Finally, the last block of the flow is a *Code Generator*, which produces an HDL description of the final architecture of the generated design. Common HDLs used for this description are Verilog and VHDL, but depending on the HLS tool and on the target FPGA device they may contain vendor-specific directives to instantiate proprietary components.

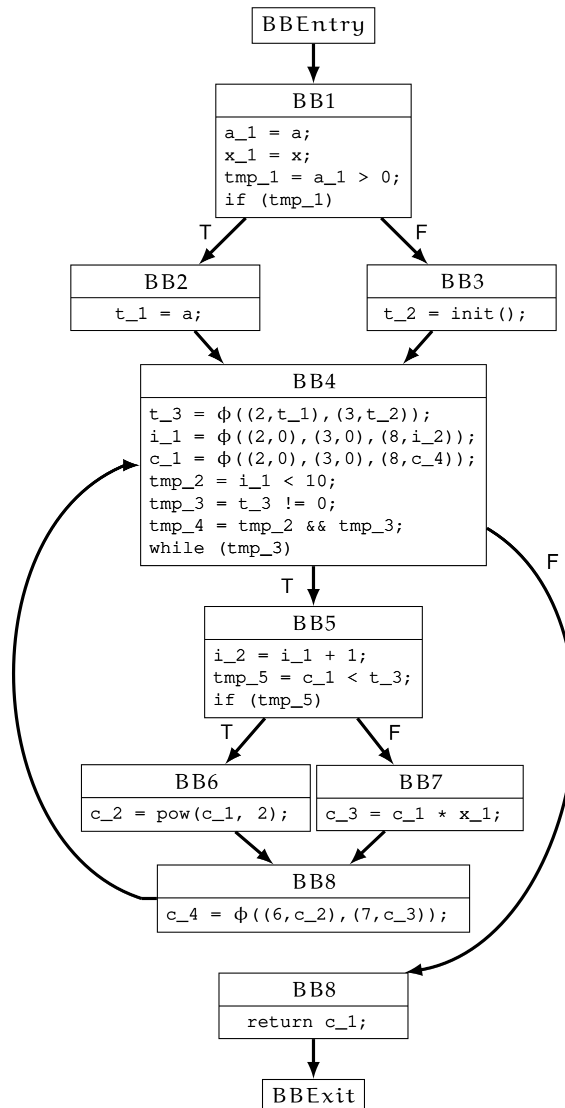
2.1.1 Front-End: Intermediate Representations, Analyses, and Optimizations

The front-end performs program analysis and transformations on the IR with the aim of producing a representation that is semantically equivalent, but more suitable for manipulation and that can improve the results of the compilation process in terms of performance and/or resource utilization on FPGA. The front-end usually builds common Intermediate Representations starting from the AST: *Control Flow Graphs* (CFG) with operations in *Static Single Assignment* form (SSA).

A Control Flow Graph [4] is a graph that represents the *control flow* of the input program. Every node of a CFG is called *Basic Block* (BB) and is composed of a sequential list of instructions, with a single entry point at the beginning and no branches or jump instructions except at the end. An example is shown in Figure 2.2, where 2.2(a) represents the original source code and 2.2(b) depicts the Control Flow Graph obtained from it. In the picture, it is easy to see that BBs

```
int f (int a, int x) {
    int i, t, c;
    if (a > 0)
        t = a;
    else
        t = init();
    i = 0;
    c = 0;
    while (t != 0 && i < 10) {
        i++;
        if (c < t)
            c = pow(c, 2);
        else
            c *= x;
    }
    return c;
}
```

(a) Source code of a C function.



(b) Control Flow Graph associated to the function f represented in (a)

Figure 2.2: Source code of a C function and the associated Control Flow Graph.

contain lists of instructions and that conditional or unconditional jumps are at the end.

There is another important thing that can be noticed in Figure 2.2(b): the operations in the Basic Blocks are not exactly the same as they are in the code in Figure 2.2(a), but they have been rewritten in another form. This is the Static Single Assignment form (SSA), which has the property that every variable is assigned exactly once in the static representation of the IR. SSA is widely used in compilers and it can be constructed efficiently for arbitrary Control Flow Graphs [20]. Without delving into the details of this process, the key idea is that the variables present in the original IR are versioned, so that operations that update a variable are converted to operations that assign a different version of the same variable. The versioning is visible in Figure 2.2(b), where the version number `n` has been appended to variable names with the suffix `_n`. The introduction of the versioning poses an additional challenge when different values are assigned to the same variables in different branches that are mutually exclusive because the values assigned in the branches must be reconciled when the branches merge together. In Figure 2.2 this happens for example with `t` and `c`. The key to resolve this issue is the introduction of the so called *phi-functions* (or *phi-instructions*, or *phi-operations*). They are represented as ϕ in Figure 2.2(b), and sometimes they are simply called *phis*. The idea is that for every BB with more than one incoming edge, if different versions of the same variable v are assigned along at least two of these edges, a phi instruction have to be inserted at the beginning of the BB to reconcile the values assigned along the different incoming paths. Every phi accepts a variable length list of pairs $((BB_0, v_0), \dots, (BB_i, v_i))$. For every $k \in \{0, \dots, i\}$, BB_k is a Basic Block identifier, while v_k is the last version of the variable v assigned on the incoming path that comes from BB_k. For example, in BB8 there is a phi instruction to reconcile versions of `c`: `c_4=ϕ((6,c_2),(7,c_3))`. This means that the value assigned to `c_4` in BB8 is `c_2` if the execution comes from BB6, `c_3` if it comes from BB7. Phi operations are virtual operations that are part of the IR, but they are not translated directly into machine code. They are usually translated into copies by compilers targeting processors, or handled with register allocation algorithms to reduce the number of copies. HLS compilers can use dedicated register allocation patterns or directly translate them into multiplexers.

The importance of SSA is that it makes easier to reason about program properties, enabling efficient implementations of data flow analyses. In the front-end of an HLS tool, all static analysis and optimization algorithms known from compiler literature can be used. These include (but are not restricted to) value range analysis, alias analysis for pointers, code motion, constant propagation, dead code elimination, common subexpression elimination, partial or total loop unrolling, and others. Some of these operations come both in intra-procedural and inter-procedural flavors. There are also some IR transformations that are not relevant for traditional compilers but that are particularly beneficial for hardware synthesis: bit-width analysis, decomposition of arithmetic operations into smaller ones, speculation and predication of operations, transformations of integer arithmetic operations with constants operators in simpler forms. All these transformations can be performed directly in the front-end at the IR level, before the beginning of the High-Level Synthesis steps.

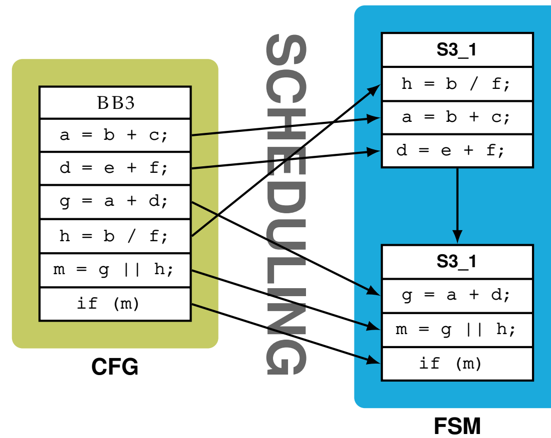


Figure 2.3: Scheduling of operations from a Basic Block to Finite State Machine.

2.1.2 The High-Level Synthesis Process

The actual High-Level Synthesis process starts from the IR generated from the front-end and is mainly composed of three steps: *allocation*, *scheduling*, and *binding*. These are actually not the only parts of the process, but they are the most important and are also highly interdependent. During these steps, the information necessary for the generation of the final hardware design is computed. In particular, from the CFG of every function of the original high-level program, the HLS process generates a component composed of a *Finite State Machine* (FSM) and a *DataPath* (DP). It may be possible to generate different architectures, for example for streaming computations, but the work described here assumes that the High-Level Synthesis flow generates an FSM and a DP. The methodology proposed here may not be applicable in other cases.

The *scheduling* step analyzes all the operations and the Basic Blocks of the CFG and it builds a *State Transition Graph* (STG) that models the FSM of the hardware component that will be generated from it. In the rest of this work, the terms *Finite State Machine* and *State Transition Graph* are often used as synonyms. To generate this representation, every Basic Block is divided into *control steps*, and every operation in the BB is then assigned to one of these *control steps*. *Control steps* are then grouped into *states* to generate the FSM. In practice, operations scheduled in the same state of the FSM are executed in the same clock cycle in the generated hardware. It is important to notice that multiple operations can be executed concurrently in the same state, even in *chaining*, as long as data dependencies are met. Moreover, if there are no data dependencies between operations in the same BB, they can be reordered during scheduling, like shown in Figure 2.3. The figure shows only one BB taken from a Control Flow Graph, how it is mapped on two consecutive states in the generated FSM, and how single operations are scheduled in such states. The operation `h=b/f` is anticipated in state **S3_1** because its inputs are not dependent on previous operations. This kind of reordering is frequent in *scheduling* and it aims at minimizing the cycle latency of the final circuit, while also taking into account other constraints such as resource availability determined by *allocation* and *binding*. For example, in Figure 2.3 the operation `h=b/f` could be anticipated because it has a latency of two cycles and scheduling it in state **S3_2** would require an additional state after **S3_2** to wait for its termi-

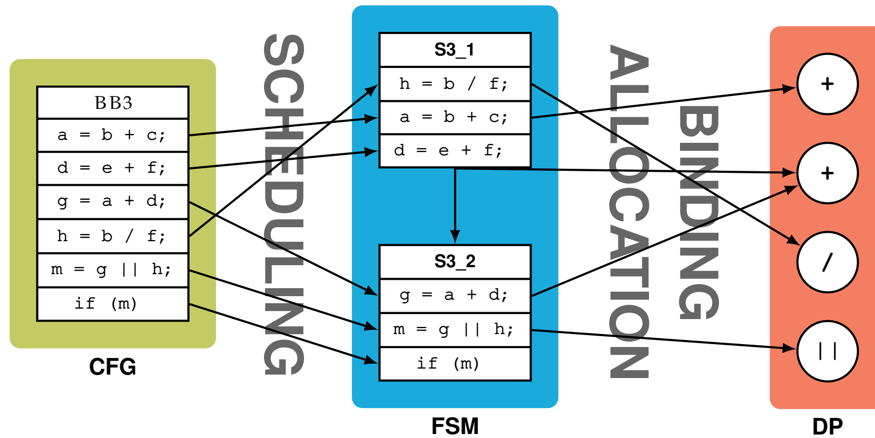


Figure 2.4: Scheduling, allocation, and binding mappings. Scheduling assigns every operation in a Basic Blocks to a state in the Finite State Machine; allocation selects the kind of hardware component in the DataPath used to implement the operation; binding identifies which specific instance of a given component type is used to execute the operation in the DataPath.

nation. Operation `g=a+d`, instead, could be postponed because it has a latency of only one cycle and scheduling it in state $S3-1$ would require instantiating an additional adder to execute it in chaining with `a=b+c` and `d=e+f`.

These are only a few examples of the optimizations that can be performed during *scheduling*. One important characteristic that is preserved during *scheduling* is that every single BB in the Control Flow Graph is mapped onto a list of sequential states in the generated FSM, with branches only at the end. This is an obvious consequence of the fact that all the jump instructions contained in a BB are necessarily at its end. This property is fundamental for the definitions of *Control Flow Traces* and of *Control Flow Equivalence* given in Chapter 5.

As explained above, *scheduling* is mainly concerned about clock cycles and timing of execution of the operations. *Allocation* and *binding*, instead, are related to the actual implementation of the operations. *Allocation* selects functional units to be used to perform the elementary operations contained into the IR. Usually, synthesis tools have a library of components to choose from. The library usually contains multiple modules that can perform an operation of the IR with different area and timing. The allocation process is generally constrained by some performance goal of the design. *Binding* assigns operations to specific instances of the functional units used to implement them, as selected from *allocation*. An example of the result of these operations can be seen in Figure 2.4, where the previous *scheduling* example is integrated with a schematic representation of components in the DataPath and of the mappings computed by *allocation* and *binding*.

In order to compute these mappings, the HLS engine has to take into account several different data for every single operation. The first necessary information is the type of component used to implement the operations, but also the latency is important. In this respect, HLS considers two kinds of operations: *Fixed Latency Operations* (FLO) and *Variable Latency Operations* (VLO). FLOs are usually small operations, with a fixed execution time of a few cycles, that is known at compile time. They usually have no side effect and they can be implemented with pipelined components to improve throughput. VLOs, instead, are used to model operations with long or unpredictable latency, like accesses to external memories,

function calls, or long operations in general, even if their latency is known. Long operations could be treated FLOs but, unless there is 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. The first is used by the caller to initiate the execution of the VLO, while the second is used by the VLO to notify its termination to the caller. This allows the caller to wait in an idle state the termination of the VLO, without needlessly increasing the size of the FSM. Other data used for *scheduling*, *allocation*, and *binding* are the sharing patterns of the components. If two operations are mapped onto the same shared component they cannot be scheduled in the same state of the FSM. Even if they are not in the same state, the execution time of the shared component must be taken into account for *scheduling*, along with the pipeline initiation time if it is a pipelined module. Different sharing patterns can also enable optimizations of the interconnection logic between different components. The same holds for memory allocation: if the alias analysis in the front-end can guarantee that operations performing memory accesses are on separate memory locations the accesses can be parallelized. All these subtleties of the mappings between operations and components used for their implementation in hardware will be reprised in Chapter 5, where they will be used for the definitions of *Op-Traces* and *Data Flow Equivalence*.

All these scenarios are just a subset of the kind of decisions that HLS tools have to make, but they help to realize how tightly dependent on each other all the HLS steps are. They also give an idea of how complex can be to debug hardware generated with HLS, especially if the designer has little knowledge of the internals of the specific HLS compiler used for the generation. Chapter 5 will show how to tame this complexity to design a methodology for automated bug detection without user interaction.

2.2 Memory Allocation and Hardware Synthesis of C Pointers

The semantic of pointers, as defined by the C language, represents the address of data in memory. This definition starts from the assumption that the target architecture consists of a single contiguous address space that contains all the data of the running application. This assumption is not necessarily valid in HLS, which means that HLS tools need to handle addresses and pointers with particular care. The reason is that, on FPGAs, data can be stored in separate memory blocks, possibly with different underlying technology. Moreover, if the address of a variable is not taken in the program, the HLS tool is allowed to perform optimization so that the variable may not even be allocated in memory, but its value is held by a register or a wire. To make all these decisions, the HLS tool have to consider if the variables are alive on the boundaries of FSM states, along with alias analysis results and scoping. Optimizing memory allocation is very important in HLS. Several different results show that it leads to significant improvements of the generated designs [8] [67] [88] [95] [89] [76] [68]. Every technique has benefits and subtleties. Thus, a flexible methodology for debugging memory allocation and pointer operations in HLS tools must be kept independent from the underlying memory technologies and allocation algorithms.

The hardware synthesis of pointers and their optimization has been analyzed thoroughly by Séméria et. al in [74] [75]. These works define the foundations of the hardware synthesis and optimization of pointers in the current HLS methodologies. In [75], the authors analyze the operations that can be performed through pointers in the C programming language and propose strategies for their synthesis. The three operations taken into account by this work are loads, stores and dynamic memory allocation and de-allocation. The authors distinguish between two situations for the treatment of loads and stores: pointers to a single location and pointers to multiple locations. Pointers to a single location can be removed replacing loads and stores operation with equivalent assignment operation to the pointed object. In the case of pointers to multiple locations, load and store operations are replaced by a set of assignments in which the location is dynamically accessed according to the pointer value. Addresses (pointer values) are encoded using two fields: a *tag* and an *index*. The *tag* is used to represent the memory module used to access the data. The *index* is an offset to specify the precise location of the accessed data inside the memory module represented by the *tag*. With this strategy, loads and stores can be removed using temporary variables and branching instructions. In [74], the focus is on the analyses and transformations that can be performed in order to optimize the synthesis of loads and stores operation through pointers. The dynamic memory allocation and de-allocation through `malloc/free` are addressed using both hardware and software memory allocators. [75] concentrates on the hardware solution giving hints on a possible software strategy. The proposed solution makes use of a hardware module implementing the `malloc` and `free` functions of the `C` standard library. The proposed module is able to allocate storage on multiple memories in parallel. As a consequence of this design choice, the dynamically allocated space is partitioned into memory segments.

In general, for memory allocation, HLS tools take two decisions:

1. which variables have to be stored in memory, typically arrays, `structs`, global variables, and variables with `static` storage or `volatile`, but other can be placed in memory as well;

2. the location where every memory-mapped variable is stored, including different partitioning schemes and position in the memory hierarchy in the final architecture generated by HLS.

The first point is usually inferred using *alias analysis* (or *points-to analysis* [80]) and/or decided with explicit directives. The second option is tightly bound to the HLS implementation, to the specific memory architecture of the generated design, and to the set of performed memory optimizations.

For every given combination of these things, the debugging technique proposed in this thesis (Chapter 8) makes a general assumption: every variable must be mapped to a specific memory location. In this work, a *Memory Location* is an unambiguous position in the generated hardware. It can be described with a unique identifier for the memory module, a position in that memory module and the size of the object stored in that position. This holds for any underlying memory technology, being it a ROM, a BRAM, or an external DDR. With this assumption every memory-mapped variable `i` is associated with a *Memory Location*.

A *Memory Location* can be defined as a triplet $\langle M_i, B_i, S_i \rangle$, where M_i is a unique identifier for a memory module (independent of the memory technology), B_i is an offset in bytes in that memory module, and S_i is a size. In general, the size S_i must be expressed in multiples of the memory alignment. In the rest of this thesis, *Memory Locations* are assumed to be byte aligned, but any other alignment can be used without loss of generality.

This concept of *Memory Location* is similar to the *location sets* introduced by Wilson and Lam [91] and also used by Séméria and De Micheli [74]. In compilers, alias analysis works on pointers, trying to recover the set of memory locations where they can point to: the *points-to set*, or *location set*. The results of the analysis can be used to tune the hardware memory partitioning. Notice that the *points-to sets*, like the *Memory Locations*, are abstract concepts, independent of the target architecture. Hence, the same strategies described in [75] and [74] can be used for hardware synthesis and optimization of addressing logic.

The only thing necessary for the implementation of the Address Discrepancy Analysis described in Chapter 8 is that it must be possible to identify the signals representing the addresses in hardware using HLS information. The values of those signals will be used by the Address Discrepancy Analysis algorithm. Like in [74], M_i is an abstract identifier, that may actually not be translated in hardware, depending on optimizations and static analysis. In particular, when a memory operation can be attributed to a single local/private memory module, M_i can be completely optimized away. In cases where M_i is not translated to hardware, only B_i and S_i can be used for *Discrepancy Analysis*. When B_i and S_i are optimized away, M_i is sufficient to retrieve the *Memory Location*.

The proposed approach aims at handling complex memory allocation patterns. To support array partitioning, the assumption must be slightly restricted. Requiring a variable to be mapped on a single *Memory Location* is clearly not enough. Instead, the assumption is restricted to require that *every element* of the array itself (or field in a `struct`) is associated with a single *Memory Location*. To summarize, this means to be able to compute the inverse function of the mapping of high-level variables onto hardware *Memory Locations*. In case of arrays and `structs`, this inverse function needs to have per-element granularity. In the rest of the discussion on automated bug detection on `C` pointers, the term 'variable' will be used loosely, with the meaning of 'scalar variable or field in a `struct` or element of an array'. The goal is to avoid to weigh down the terminology during the discussion of what will be called *Address Discrepancy Analysis*. In this way, the discussion can be kept general while focusing on the approach, treating all the data types in the same way.

2.3 High-Level Synthesis of Multi-Threaded Programs

The debugging methodology described in the thesis was initially conceived to debug circuits generated with HLS from single-threaded programs. However, Section 7.3 shows how it can be extended to handle hardware and software multi-threading. For this reason, this section gives an overview of the methodologies and the approaches used to support parallel programming languages in HLS.

As FPGAs become more competitive for the acceleration of parallel workloads and irregular applications, the necessity for suitable programming models grows.

High-Level Synthesis accepting multi-threaded programs as input is seen as a way to exploit the available physical parallelism while keeping the useful high-level abstractions that make HLS competitive. The current trends are investing on high-level programming languages, mostly based on the C programming language, that are already industry standards for programming parallel general purpose processors, such as CPUs and GPUs. Recent HLS tools support one or more paradigms among C extensions like OpenCL [46], CUDA [63] and OpenMP [83], or standard C libraries like `pthreads` (POSIX Threads [2]).

Efforts on OpenCL are leaded by the main FPGA vendors Xilinx: [92] and Intel [41] (formerly Altera [21]). Hosseinabady and Nunez-Yanez [35] have proposed improvements to the synthesis of OpenCL workgroups in hybrid ARM-FPGA devices. Owaida et al. [64] [65] have created a Finite State Machine with a DataPath model suitable for the 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 (Papakonstantinou et al. [66]) 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 (Nguyen et al. [62]). Cabrera et al. extend OpenMP directives to target more closely FPGA-specific characteristics [11]. They propose a `target device` pragma to instruct the compiler about the target device for the synthesis, a `label-name` clause to mark data as `input/output/inout`, and a `block` pragma to control resources in loops. OpenMP loops are supported by the BAMBU compiler [16] based on GCC [79] and by the LegUp compiler [17], based on LLVM [48], which also supports nested parallelism using `pthreads`. LegUp and BAMBU exploit physical parallelism, instantiating multiple copies of the components, one for every thread on FPGA.

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 [86] to generate multi-threaded accelerators starting from loops constructs [30]. 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. [82], 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. Huthmann and Koch [37] further explore the same idea to reduce resource consumption on FPGA, using heuristics and profile-guided optimizations to change the number of supported threads at different stages of the pipeline.

There are also other works 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. [6] 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. [47], 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 for the purposes of this thesis is that they are actually agnostic about this mappings and they are focused on providing the necessary system-level integration once the mapping has been computed.

Wang et al. [90] 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 multiprocessors System-on-Chip completely placed on FPGA, like proposed for example by Ma et al. [50]. 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.

With such a large variety of methodologies and implementations, it may seem hard to find a single technique for automated bug detection in multi-threaded hardware. The task is not simple because the HLS of multi-threaded code can take different decisions on three subjects. The first is the nature of the threads in hardware: physical or logical. Some approaches duplicate hardware components, others add logic to suspend and resume multiple logical threads on the same accelerator to hide memory latencies. The second is whether the assignment of a task to a certain hardware accelerator is decided statically at compile time or dynamically during the execution. The third is about the homogeneity of the threads, i.e. if they all execute the same high-level function (even if they may follow different branches during execution), or they execute different functions. Independently of how a specific implementation answers to these questions, to debug digital designs generated with HLS from parallel programs as described in this thesis it is necessary to make an assumption: that using HLS information and observing the right signals in the design during execution it must be possible to tell which task is in execution on a certain physical copy of a hardware accelerator. If the mapping of tasks onto components is static, this is trivial using HLS information. Otherwise, depending on the implementation, it may be necessary to use the values of signals to reconstruct this information at runtime. This may seem restrictive, but approaches based on the analysis of the discrepancies between hardware and software are intrinsically tightly coupled with HLS tools, because it is the HLS tool that defines the mapping between hardware and software. Hence, it is enough that the HLS tool exports the necessary information to the debuggers. This is actually also one of the major advantages of techniques based on discrepancy analysis because it allows them to know all the optimization performed during HLS without user interaction, making possible to spot subtle bugs that are usually easy to understand only to developers of the compiler itself.

Summary

This chapter introduced the background concepts necessary for the understanding of the thesis. Section 2.1 provided an outline of the High-Level Synthesis, its steps and the optimizations it can perform to generate optimized hardware from high-level `C` code. Section 2.2 delved into the details and the subtleties in-

volved in the process of hardware synthesis of C pointers, while Section 2.3 focused on describing the current state of the art in the field of High-Level Synthesis of multithreaded programs. The main topic of the thesis is *Discrepancy Analysis*, a methodology for automated bug detection in circuits generated from High-Level Synthesis. Two of its main features are the support for bug detection on pointers and the capability to analyze designs generated from multithreaded code. Hence, the three topics described in this Chapter will be very useful throughout the work. Chapter 3 digs more in details on different methodologies for hardware debugging, initially from a broad perspective and then focusing on circuits generated with High-Level Synthesis.

3

State of the Art of Debugging Methodologies for HLS

This chapter describes the current state of the art of debugging techniques for circuits generated with High-Level Synthesis. Section 3.1 introduces hardware debugging in general, describing the fundamental characteristics of a good debug methodology and the main categories of techniques. Section 3.2 outlines the specific challenges of debugging hardware designs generated with HLS. Section 3.3 describes different classes of state of the art debugging methodologies for circuits generated with High-Level Synthesis. The methodologies are grouped together for the similarity of the approach they use and according to how they tackle some of the challenges described in Section 3.2 and to the categories introduced in Section 3.1. A particular attention is given to the results of various different research efforts that are relevant for comparisons with different aspects of the methodology proposed later in the thesis.

3.1 Concepts of Hardware Debugging

Hardware debugging is known to be a complex and time-consuming process. Typically it involves selecting a large number of signals in the design, tracing their values concurrently during the execution, and analyzing them to find misbehavior. To do this effectively, hardware debugging technique needs to provide three main features [42]

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

Signal observability is the ability to observe the values of the largest number of signals, registers and memories in the design, with the finest granularity, across the largest time span as possible. This is necessary to actually see what is happening in the design during its execution.

Hardware controllability is the fine control on the design execution during the debug operation. It is necessary to detect not only the wrong signal but also the exact time when that happens and possibly the values of a number of other signals in a surrounding time frame.

Turnaround time, in general, means the time between a request to the debugger and the attainment of the result. This time must be short enough not to slow down the whole development and verification process.

Achieving these goals usually requires different kinds of trade-offs between the accuracy of a debugging methodology, provided by observability and controllability, and the resulting turnaround time.

3.1.1 Simulation-Based VS On-Chip Debugging

Research efforts and state of the art practices in this field can be subdivided mainly into two groups, both with intrinsic advantages and drawbacks:

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

Debugging *on-chip*, also called *in-circuit*, means synthesizing the RTL design and running it on an FPGA while observing its behavior. This is the only way to spot malfunctions due to hardware faults (power-supply noises, environmental interference, damaged gates). However, for logical bug detection, in-circuit debug is usually worse than simulation in providing observability and controllability. Indeed, to guarantee these two properties it is necessary to insert additional control or tracing components in the design or to enforce restrictions on the memory layout of the hardware accelerators. This does not scale with increasing design complexity and it also imposes limits on the number of traced signals and on the time frame that can be captured. The main reason is that logic and memory available on an FPGA are limited by the device. The extra logic or the forced memory layout may also modify the original design, compromising crucial timing characteristics of the accelerator, or making the bug impossible to reproduce. Even when the insertion of the debugging components is harmless, finding and analyzing the interesting signals can be a hard task and requires deep knowledge of the HDL and the underlying electronics characteristics of the target technology. Another problem of in-circuit debugging is related to the turnaround time. Debugging is an iterative process by its nature, and it usually requires several trials and errors. If users have to modify the hardware debugging components or change the place where they are inserted at each iteration a lot of time is wasted just for these operations instead of focusing on the analysis of the hardware behavior. reconfiguration of the bitstream is always necessary.

On the other hand, *simulation-based* debugging involves simulating the design under test on a host machine, without actually running it on FPGA. In general, simulation is far slower than hardware execution, but this does not inevitably lead to longer turnaround times. Indeed, the simulation takes much more time than real hardware execution, but incremental trials and errors can be done during the debug process without altering the design. No extra circuits or dedicated memories have to be inserted to provide signal observability since area and memory limits on FPGA are not an issue with simulation. Controllability can be guaranteed using simulators' Application Programming Interfaces (API), enabling to execute the design until a determined point, stop it, analyze variables and then resume the execution (or even roll it back) without affecting the logic of the simulated circuits. If the values of the signals are dumped to a file there is even no need for controllability at all, because the hardware can be simulated and the values inspected later. In this way, simulation succeeds in achieving complete observability of the signals, while keeping reasonable turnaround times.

3.1.2 Online VS Offline debugging

There is another distinction that can be made for hardware debugging methodologies. The techniques can be subdivided according to the actual time when the debugging operations happen, compared to the execution or the simulation of the circuit. The two categories are following:

- *Online debugging* - the behavior of the design under test is observed directly during its execution;
- *Offline debugging* - the behavior of the design under test is recorded in some way during the execution and analyzed later.

Notice that these categories are independent of and orthogonal to the distinction between simulation-based and on-chip methodologies described in Section 3.1.1. This means that there are techniques for both online and offline on-chip debugging.

Offline on-chip debugging usually requires dedicated components to register execution traces during the execution of the circuit. These traces are collected in some memories or sent directly off-chip and analyzed in a second stage. *Online on-chip* debugging, instead can involve additional components that are able to check assertions on the fly and/or some control logic that allows halting the circuit during its operation to analyze values in registers and memories. However, this last strategy may potentially disrupt the interactions with other parts of the system, making impossible to continue the execution of the circuit and potentially introducing other bugs.

There are also techniques for online and offline debugging based on simulation. *Offline simulation-based* debugging consists of collecting execution traces during the simulation, in some waveform format. The waveforms are then analyzed manually to find bugs. *Online simulation-based* debugging, instead is possible through simulators' Application Programming Interfaces (API). The user can interact with the simulation kernel, asking to inspect values of signals, memories, and register during the operations. With some simulators, it is also possible to artificially change the value of a signal. All these things are not possible or do not allow the same flexibility with online debugging on-chip, because they would need dedicated debugging components or because the underlying technology simply does not allow it. In this respect, online debugging based on simulation provides a workflow that is more similar to software debugging.

3.2 Challenges in Debugging Hardware Generated with HLS

Section 3.1 describes hardware debugging from a broad perspective, helping to understand some of the fundamental concepts. It introduces ideas and categories necessary to the discussion, but it does not focus on High-Level Synthesis and the specific challenges that arise in case of bugs in HLS-generated circuits.

In traditional hardware design, engineers write HDL code and are very well aware of the quirks of the digital electronics that will run on FPGAs. This means that in case of bugs in the circuit they have most of the information that is necessary for debugging. Debug is known to be a complex process, but it can be easier if who debugs the circuit is also who designed it in the first place because he or

she is usually more familiar with the design. Even in this case, the process typically involves various trials and errors to understand what are the parts of the circuits involved in the malfunction, and especially to pinpoint the original root cause. The behavior of the design must be explored manually and this is very time consuming and error-prone.

With the introduction of HLS, all these issues are exacerbated. On one hand, the design process is improved: it is faster and it requires less expertise in hardware design so that even software engineers can contribute. The complexity of the HDL and of the underlying digital electronics are masked by the HLS layer. On the other hand, these same things make debugging harder: the designers may not be proficient in HDL, the generated designs are not necessarily intended to be human-friendly, and given that they were not written by hand it is hard for a designer to understand what may be the cause of a bug. Things are even more complicated by the fact that modern HLS tools are able to apply a wide range of optimizations. These same optimizations are what makes possible to obtain very good results in HLS, but they can heavily modify the circuit, making even more complicated to correlate the HDL with the original high-level source code. Moreover, the only possible way to accurately reconstruct this relationship is to access all the information available to the HLS engine, and to have a deep knowledge of the internal of the used HLS tool. This is actually not realistic because, except for academic open source software, it's extremely rare that HLS users have access to the HLS internals.

An additional degree of complexity is introduced by hardware synthesis of pointers and by HLS of multi-threaded programs.

Pointers and addresses complicate debugging because there is no direct mapping between the single address space in software and the memory layout in the generated hardware. The reason is that different variables are usually allocated in physically separated memory modules on FPGA. Like explained in Section 2.2, every module has its own addressing logic that can be optimized in different ways depending on the memory allocation step of HLS. Practically, this means that in order to understand what is actual object allocated in a memory module, and to understand the real meaning of an addressing signal passed to a memory unit, it is necessary to exploit HLS memory allocation information.

Threads increase the complexity in another dimension. Like explained in Section 2.3, they can be implemented in different ways by HLS tools, either exploiting physical parallelism or using logical parallelism to hide latencies. Despite the approach adopted by an HLS tool, this means that the architecture of the component generated from multi-threaded programs is different from those generated from single-threaded programs. Developers have to be aware of the mapping between software and hardware threads to perform accurate debugging. Again, this means that they need to have access to information that is only available to the HLS tool. The more advanced are the multi-threaded hardware implementation, the highest performance can be squeezed out of the same FPGA and, at the same time, the hardest is to debug the designs without HLS information.

In this scenario, it is possible to identify the main challenges of debugging hardware generated with High-Level Synthesis, compared to digital circuits designed with more traditional methodologies. They all stem from the two folded nature of HLS: on one hand the high-level abstraction simplify development and

enable the tools to perform smart optimization; on the other hand, the masked information is vital for effective debugging and must be brought back to developers when debugging, without necessarily requiring HDL expertise or knowledge about HLS internals. For these reasons, an effective methodology for bug detection targeting HLS-generated designs must necessary deal with the following challenges.

- Reduction of user interaction, to avoid the need of HDL expertise or exposure to HLS internals from users.
- Automatic identification in the generated design of the signals necessary for debugging.
- Backtracking of the relationships between high-level source code and generated HDL.
- Supporting front-end compiler optimization at the level of Control Flow Graph, with observability of temporary variables introduced by the compiler.
- Supporting architectural optimization of the generated designs, like chaining, pipelining, and sharing, but limited to them.
- Supporting for different memory layout and partitioning, while providing support for automatic HW/SW address space translation, to enable debugging pointers and addresses.
- Supporting different schemes for HLS of multi-threaded programs, while retaining all the previous points in this list.

Part II of the thesis is dedicated to the definition and the implementation of a methodology that addresses these points. The proposed technique tries to provide all these features, with a debug workflow for HLS-generated circuits that relieves the users from the burden of manual debugging, leaving the time-consuming task of bug detection and isolation entirely to the machine.

3.3 Debugging Methodologies for High-Level Synthesis

In recent years, many results have been pushing the limits of the debugging capabilities for electronic circuit designs generated by HLS tools. Industrial High-Level Synthesis frameworks now integrate environments for verification and debugging [81] [61] [53]. The idea they all have in common is to take advantage of the additional information present in the original high-level specification and to use it improve debugging. In addition to what said in Section 3.1 about hardware debug methodologies, the approaches targeted to HLS-generated designs can be roughly grouped into three main classes.

1. Approaches focused on architectural support for the debugging of components generated with HLS. These techniques use HLS information for automatic and efficient implementation of debugging components. Such components are embedded in the designs and can provide observability, tracing the behavior of the circuits during execution, of controllability allowing users

to suspend the execution on the fly. In general, these methodologies provide the building blocks for the others and are often composable with them: an optimized component that enables debugging operations can be used by other methodologies to achieve higher level goals and implement innovative debug flows.

2. Approaches focused on providing a software-like user-friendly debugging experience. These methods provide to users the means to insert breakpoints, to observe values of signals, and to perform a variety of operations that are familiar to software developers. To this end, they either rely on simulators' API or they use dedicated component to do it directly on-chip. Some of these approaches also show the relationships between the HDL and the original source code, but this is possible only with variables that were already present in the original source code and not with those inserted by the compiler optimizations. These means that the more compiler optimizations are active, the less these methods are accurate and useful.

3. Approaches based on the instrumentation and execution of the Intermediate Representation (IR) of the HLS compiler or of the original source code, which is used to generate a golden reference for the behavior of the circuit. This golden reference is then used to check automatically if the design is working properly at a functional level, and to report useful source-level information to the users. This last class of techniques has seen a wide number of contributions in the past few years. The *Discrepancy Analysis* described in this thesis belongs to this family.

3.3.1 Optimized Component for Architectural Support of On-Chip Debug

An example of the first type of approach has been proposed by Monson and Hutchings [58] [59], which use source level transformations to insert tracing logic (Event Observability Port and Buffers) for the output signals of operations. The architecture of these components is tuned using HLS information about the Control Flow Graph and the Finite State Machine, to maximize the number of events that can be registered in a buffer. In [58] and [59] the authors used simulation for their evaluation of the results. This methodology has been ported on-chip in a subsequent work [57]. However, the approach does not focus on the Finite State Machine (FSM) used to control the circuit, but only on the DataPath, and is thus not suitable to detect bugs involving control flow. The challenges of how to maintain the relationships between high-level source code and the generated hardware are not in the scope of the methodology. In addition, given that the trace buffers are per-signal, additional processing is required to reconstruct the time relationships between the traces. This becomes very hard when the necessary variables are not stored in a register, or even impossible when heavy compiler optimizations are activated.

Goeders and Wilton [26] [29] generate a component aimed at managing the debugging operation, and at saving the execution traces on FPGA. They show different techniques to reduce the memory usage necessary to store the trace at runtime, and to support compiler optimizations [27]. In a successive work, the authors also adapt the architecture of their components to debug circuits gener-

ated from multi-threaded programs [28]. In this latter work, the hardware accelerators used to implement each thread share a single debug component used to store all the execution traces in a compact way. However, due to the nature of the compression they implement, this approach does not give significant benefits in case of homogeneous threads, exploiting physical parallelism to execute multiple instances of the same task. The reason is that the sharing of the component used to store the traces generates contention and slowdowns when there are concurrent accesses to save traces from different threads during the same cycle. Hence, the methodology is not beneficial in case of homogeneous threads, like for example OpenMP for loops or `pthreads` executing the same function. This is in contrast with the current trend of starting HLS from these input specifications.

3.3.2 Software-Like User-Friendly Debug Flows

The goal of Goeders and Wilton [26] [29] is also to build a higher level debug framework, that provides a user-friendly debug flow. The components they describe are used to perform on-chip debugging both online and offline, with different modes. Users can manually inspect the traces after execution or can suspend the hardware to analyze its state. Unlike what is proposed in this work, they do not provide the automated selection of the signals necessary for debugging, but it is up to users to decide what has to be analyzed by the debugger. For what concerns online debugging, they also note that suspending the execution may break interactions with other components of the system and potentially introduce other bugs. For this reason, in multi-threaded hardware the analysis of the traces is performed offline [28].

Other works try to provide different flavors of software-like debug flows. One of the first works to discuss it is from Hemmert et al. [34], but it does not describe an actual implementation.

Calagar et. al [12] tackled this limitation, bringing to HLS some of the typical debugging operations: stepping, breakpointing, and dynamic variable inspection. Their debugger, *Inspect* takes a double approach: both on-chip and simulation-based. For on-chip analysis, the authors use Altera SignalTap leading to high memory usage, as also reported by [26]. For simulation-based debugging, they run at the same time a simulation of the generated design and the binary obtained by the original source code in a debugger. Using APIs of the simulator and of the software debugger, they are able to insert breakpoints, watchpoints and provide source-level information to the users. Unfortunately, this approach suffers from one of the known limits of software debugging: the temporaries introduced by compiler optimizations cannot be inspected and the accuracy decreases with the increase of the optimization level. Moreover, several other constraints are imposed to High-Level Synthesis. Namely, inlining is disabled, local RAMs are made global, and constants cannot be stored on ROMs.

3.3.3 High-Level Synthesis of Assertions

Another trend in on-chip debugging is based on hardware synthesis of ANSI-C assertions, that are translated into assertion checker circuits. This group contains methodologies that try to merge ideas from the two previous groups: workflows focused on providing software like debug environments, and techniques to

provide architectural support for debugging HLS-generated circuits. Performing HLS of assertions requires, on one hand, to design the architecture of the assertion checkers components, and on the other hand, to provide users the useful high-level abstraction of source-level assertions.

In the work of Ribon et al. [71] the checkers are synchronized directly by the checked FSM, making impossible to trigger the assertion if the accelerator enters in a hanging state. In other works, the checker's FSM is executed concurrently to the controlled module. Curreri et al. [19] duplicate shared data between checkers and checked FSM, to avoid conflicts, leading to large memory consumption. Ben Hammouda et al. [9] [31] [32] describe and implement a complete HLS flow using HLS information for the automated construction and insertion of checkers with considerably small footprint. The methodology is effective, but the main problem of assertions is that they can only check malfunctions foreseen by the developers. The assertion must be manually inserted in the original `C` specification. This fails to spot bugs that are not guarded by assertions. Finally, if the circuit happens to enter in a hanging state, the relevant assertion trigger point may be never reached at all. In addition, none of the discussed works on assertion-based verification for HLS mentions support for coarse-grained parallel programming paradigm.

HLS of assertion checkers is also used for Runtime Verification. Selyunin et. al [73] use HLS to generate runtime verification checkers in automotive chip design. Runtime Verification differs from the automated bug detection proposed in this thesis because it generates checkers for temporal logic properties that must hold for all the possible executions of the circuit. Another difference is that in runtime verification the properties to be checked must be specified in some way by the designer. Runtime verification checkers guarantee that some properties hold during all the lifetime of the checked system, and they are even embedded in final products. The approach described here, instead, helps HLS developers to efficiently and accurately find bugs, without the need of specifying temporal logic properties and automatically backtracking bugs to the original source code.

3.3.4 Automated Bug Detection

All the results mentioned until now still leave aside most of the challenges outlined in Section 3.2. In particular, they still leave to the designer the burden of finding out the interesting signals and to step through the execution to find bugs. These tasks can be overwhelming with increasingly larger designs. Things are even more complicated if the circuit description is generated by an HLS tools because the developer has no knowledge of the signal naming conventions and how the signals are related to high-level variables.

From these needs has arisen an entire class of debugging methodologies, that focus on providing automated bug detection in circuits generated with HLS. These approaches use the software obtained from the original high-level code as a golden reference for the behavior of the generated hardware. The software or the front-end IR after optimizations is instrumented and executed to generate the reference execution traces. These traces are then used in different ways to ensure that the generated hardware exhibits an equivalent behavior. Common approaches either rely on simulation to generated hardware traces that are then compared with software, or they use the software traces to create and integrate

dedicated hardware checkers in the generated circuits. The challenge in this latter case is to ensure that the checkers do not alter the execution of the design.

Approaches for automated bug detection have grown considerably in recent years. The methodology described in this thesis belongs to this family. Part II introduces the methodology, while Part III shows experimental results and comparisons with other works in the same family. Here, such approaches are summarized to provide an overview for the reader. This is useful to follow the rest of the discussion and to have an idea of how the methodology proposed in the thesis relates to others in the same field.

Campbell et al. [14] 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 from the circuit, which they call hardware signature. The golden reference and the hardware signature are then compared at the end of the execution and bugs are automatically detected. Campbell et al. [13] use the same methodology on FPGA, but differently from [14], they rely on simulation for the generation of the hardware signatures. Yang et al. [93] [94], 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 [13], 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. The same does Carrion Schafer in [72]. On the contrary, Calagar et al. [12] 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 they use a conventional debugger to observe the software on the fly, the Application Programming Interfaces of the simulator 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 [58]. Iskander et al. [43] propose an approach composed of 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, to insert breakpoints and to build an environment for a software-like debugging experience.

3.3.5 Debugging Designs Generated with HLS from Multi-Threaded Programs

Hardware designs generated with HLS from multi-threaded programs have not received the same attention. Despite the innovations in the field of hardware synthesis of threads, most efforts are still only focused on how to make it possible instead of how to make it practical to debug. Besides the work of Goeders et al. [28] mentioned in Section 3.3.1, one of the few contributions is a work by Verma et al. [85], targeting OpenCL for FPGAs. The authors describe open-source debug components, modeled both in the OpenCL language and in Verilog [38] Hard-

ware 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 technology 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 compare 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 hardware component executes that task. On the other hand, 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 accounts by existing discrepancy analysis approaches. In [28] and [85], 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 necessarily deterministic. The approach described in this thesis belongs to the category of traced-based discrepancy analysis techniques, but it also targets thread parallel programming models, trying to tackle these problems.

Summary

This chapter started focusing more closely on the specific research fields of the thesis. Section 3.1 described general concepts of hardware debugging, providing a classification of debugging methodologies and highlighting their strengths and weaknesses. Section 3.2 then restricted the discussion to the debugging of hardware generated with High-Level Synthesis, explaining the challenges that HLS introduces on top of the typical problems of hardware debugging. Finally, Section 3.3 provided a broad review of the most recent advances in research on debugging and automated bug detection for HLS-generated hardware designs. This discussion gives a picture of the current state-of-the-art of the field and sets the frame for the definition of *Discrepancy Analysis* is Part II.

PART II

METHODOLOGY

This part introduces and describes a methodology for automated bug detection in hardware designs generated with High-Level Synthesis, called Discrepancy Analysis*, which constitutes the fundamental contribution of the thesis.*

Chapter 4 introduces the problem that Discrepancy Analysis *is set to solve.*

Chapter 5 provides a definition of equivalence between hardware and software executions. This definition is then used throughout the rest of the work to decide if there is a mismatch between the execution of a hardware design generated with High-Level Synthesis and the software it is derived from.

Chapter 6 describes two different workflows for automated bug detection based on Discrepancy Analysis*:* Simulation-Based Offline Discrepancy Analysis*, and* On-Chip Online Discrepancy Analysis*. These two approaches show that the methodology is flexible and adaptable to different scenarios.*

Chapter 7 focuses on the details of Simulation-Based Offline Discrepancy Analysis*, describing how to implement it and how it can handle both single- and multi-threaded input specifications.*

Chapter 8 extends it to handle `C` *pointer and address arithmetic.*

Chapter 9, instead, gives a complete and detailed picture of On-Chip Online Discrepancy Analysis.

4

Problem Statement and Goals

This chapter describes the motivation of the Discrepancy Analysis methodology proposed in the thesis. Section 4.1 outlines the problems that Discrepancy Analysis is set out to solve, along with the shortcomings of previous state-of-the-art that has to be overcome. Section 4.2 introduces the principal ideas and insights that led the choices for the design of the methodology. Starting from these ideas, and form the limitations of other approaches, Section 4.3 describes the goals of the work described in the thesis. Finally, Section 4.4 discusses the different classes of bugs that the approach presented in this work aims to detect.

4.1 Description of the Problem

Most of the approaches described in Chapter 3 focus on methodology for enabling users to debug digital designs. This is certainly useful and it can improve the development workflow for traditional hardware design processes. However, leaving the burden of manual analysis of the hardware entirely to users poses some additional challenges and complications when the designs are generated with HLS, as explained in Section 3.2. The single main reason of these complications is due to the nature of the High-Level Synthesis process: hiding complexity and architectural details from designers. While this is good in general for development, it makes debugging more complicated, because that very same complexity and architectural details have to be understood to detect bugs effectively. High-Level Synthesis tools take a lot of intertwined decisions to generate the optimal architecture for a given high-level specification. These decisions may involve multi-objective optimizations and design space exploration, and sometimes even small changes to the original source code can result in consistent modifications to the generated architecture. It is not reasonable to expect that users understand or reverse-engineer all the steps of the HLS process in order to be able to debug the circuits that it generates. Actually, sometimes it is not even possible. Mainstream HLS tools are closed source and their licenses are very restrictive about reverse-engineering. Moreover, hardware designers are not necessarily aware of how HLS works and of the optimizations it can do. Finally, if HLS users are simple software engineers that want to exploit FPGA computing resources they may not even be able to understand HDL.

4.2 Fundamental Ideas of the Approach

In this scenario, it is clear that users have to be helped with development tools. The key idea is the following: HLS tools hide complexity to users during the design process; given that users cannot manage this complexity on their own for

debugging purposes, it is necessary to devise a methodology that has complete access to the HLS information and that is able to handle it instead of users. One of the trending ideas is *automated bug detection*, and the *Discrepancy Analysis* described in this thesis falls in this category. They rely on the fact that if the hardware presents a bug it does not behave like the original specification, which is constituted by the original high-level source code. The workflow adopted in all the approaches based on automated bug detection is roughly the same. Initially, the software obtained from the original source code (or some form of executable IR after optimizations) is executed on a given input, to extract the golden reference for hardware execution. Then this golden reference is compared in various ways with hardware execution, to detect bugs in a completely automated fashion using HLS information on the relationships between hardware and software.

This idea is extremely powerful because it delegates to the machine all the work that it is entitled to do best, while providing to users only the final results in a form that they can understand and reason about. This is actually feasible thanks to the fact that the original source code represents the specification for the generated hardware and it can be used to generate a golden reference to evaluate the correctness of the circuit. All the necessary information is already present in the HLS IR, across all the different steps of the compilation process.

4.3 Objectives, Goals, and Features

The concept of automated bug detection based on comparison of hardware and software behaviors has inspired many efforts in recent years, as shown by the discussion in Section 3.3.4. Most of these works were not available when the effort behind this thesis was started, and they have evolved concurrently finding similar answers to the same problems. However, this thesis presents original contributions even compared to all the others approaches for automated bug detection:

- it provides a detailed description of the model used to determine the equivalence between software and hardware execution;
- it details this equivalence at two independent levels *Control Flow Level* and *Operation Level*, which represent two fundamentally different ways the hardware execution can differ from software;
- it keeps the whole model of equivalence independent of the encoding of the execution traces used for the comparison, and agnostic about how these traces are actually collected;
- it also keeps the approach independent of all the front-end and architectural optimizations that are performed during the High-Level Synthesis process;
- it demonstrates that the same methodology can be effectively used on-chip and with simulation, implementing two different workflows based on the same model;
- it explicitly explains how this model can support the analysis of discrepancies on addresses and pointers, despite the intrinsic difference of hardware and software address spaces;

- it extends the approach to support the debug of circuits generated with HLS from high-level multi-threaded program specifications;
- it significantly reduces the memory footprint for debugging components when debugging on-chip, while providing at least the same level of accuracy as with simulation and with negligible effects on the achievable frequency.

In addition, there are a number of useful features that are part of the methodology and that make the design/debug cycles more productive. These features can be considered as a byproduct of the approach proposed here because they are all necessary for its implementation. However, they also contribute to the overall improvement of the debugging experience. These features are the following:

- automatic selection in the design of all the signals necessary for debugging, relieving users from their identification;
- no direct interaction with the user during the debugging operations, making the methodology suitable to be used in continuous integration servers and extensive regression testing;
- preservation of the information on the relationships between the original high-level source code and the generated HDL, that is shown to users with useful details if a discrepancy is found.

Clearly, all the points mentioned in this section cannot be achieved in practice without relying on the internals of the HLS framework used for the implementation, but the approach described in the rest of the thesis is generic enough to be applied to any HLS tool.

Another thing to stress is that, even if at first sight the approach can resemble equivalence checking, it does not try to guarantee formal equivalence between the high-level source code and the generated hardware. Rather, for a given input set, the goal is to extend the granularity of debugging operations to find bugs at every level in the component hierarchy. The goal is to identify the exact time of malfunctions and to isolate the faulty operation/component. This allows validating the HLS engine as well. *Discrepancy Analysis* cannot guarantee formal equivalence because it works on an input test set. What it does, instead, is, given an input triggering a bug, to detect the misbehavior automatically, selecting all the necessary signals to reach per-operation granularity and providing useful information on the location and the cause. A topic that was not investigated in this work is how to generate test inputs to increase the coverage. This would make possible to reduce the chances that, after using *Discrepancy Analysis* a bug is still lurking in some hidden corner of the code. However, this is actually an orthogonal problem per se, especially because there is no straightforward relationship between coverage metrics for high-level languages and for HDLs. The topic is extremely vast and it would deserve a separate analysis.

4.4 Detected Classes of Bugs

The classes of bugs that the proposed approach aims to find are strictly related to how the "correct behavior" is modeled in all the methodologies based on discrepancy analysis. When using High-Level Synthesis, the original high-level source

code is considered as the specification, i.e. it represents the intended behavior of the circuit to generate. This intrinsically means that the input code is always considered to be "correct", hence, by common sense, it cannot have bugs. As a consequence, one might be induced to think that if behavior of the generated hardware does not match the original specification it must be due to a bug in the HLS compiler. This intuition is partially true, but it does not encompass the full complexity of the situation, and neglects large and important classes of bugs that the methodology described in this thesis is able to detect. Specifically, it is necessary to make some considerations on the fact that HLS is increasingly used for system integration with third-parties components and around the intrinsic differences between hardware and software execution semantics.

Clearly, one of the main goals of the methodology proposed here is to catch bugs introduced by HLS tools. This is increasingly useful for developer of HLS frameworks, because the implementation of aggressive optimizations to squeeze the most performance out of the smallest designs can often lead to malfunctions that are hard to detect. Some of these bugs can go undetected and be shipped to users if there are no tools to identify them. Among other characteristics, the approach described in this thesis is a valuable tool for testing and improving the quality of modern HLS toolchains. However there are other classes of bugs that this methodology is designed to catch, making them relevant and beneficial also for hardware designers that have no access or knowledge about the internals of HLS tools.

First of all, the use of HLS for system integration is becoming increasingly popular, because it helps designers to manage complexity. In this scenario, hardware designers create complex SoCs designs using a mix of HLS, hand written HDL, and IP components provided by third parties. This is good for the industry and improves productivity boosting component reuse. However, it also introduces an entirely new class of bugs, caused by errors in the integration of the components, or by faulty components that have not been thoroughly tested in the same conditions as where they are being reused. Moreover, most modern HLS tools allow to integrate black boxes component, providing libraries of IP blocks used to implement common functionalities. If something goes wrong in this process, the designers are left with little clue of what the problem could be. The situation is also aggravated by the fact that they know little about the reused components and about the external libraries of IP blocks. Therefore, this is an important class of bugs that must be handled properly in order to create a user-friendly experience for bug detection coupled with HLS.

Secondly, the fact that hardware and software have intrinsically different execution models and semantics poses two additional challenges: the first is related to *Undefined Behavior* and *Unspecified Behavior* in `C`, which is the main input language for HLS tools; the second concerns post-synthesis bugs, that are unique to hardware execution and cannot be observed in software.

Two of the two main design goals of the `C` language were performance and ease of implementation. For this reason, the `C` language specification leaves the semantics of its constructs *undefined* or *unspecified*. The subtle difference between *unspecified* and *undefined* is that the former defines a set of valid behaviors associated to a construct, leaving the choice of which to actually implement. The latter, instead, does not define any precise semantics, therefore an implementa-

tion is considered standard-conforming independently of what it does when it encounters an *Undefined Behavior*. This allows implementors to choose the most efficient implementation on a given platform. At the same time, a recent study by Memarian and Sewell [52] have demonstrated how C developers have started to silently rely on deterministic '*reasonable*' behaviors of programs containing *unspecified* and *undefined behaviors*. The reason is that in most cases most compilers always do the same '*expected*' thing, but there is no guarantee of compatibility between different compilers, or between different versions of the same compiler, or even between the exact same compiler with different optimization flags. This issue is well-known among low-level C programmers but, given that it is part of the standard, there is very little to do solve it. This is exacerbated when such C programs are translated to HDL using HLS, because this translation is not expected to preserve the semantics of *undefined* and *unspecified behaviors* that developers are silently relying upon. Actually, to run code of FPGA, most of the obvious choices for *undefined* and *unspecified behaviors* in software can have entirely different outcomes, leading to inconsistent hardware implementations. These issues are always puzzling for designers, and very hard to backtrack to a root cause. The methodology described in this thesis aims at finding these bugs as well.

For what concerns post-synthesis bugs, the issue is well known in the field [36] [54]. Obviously these bugs only affect hardware designs, not software. For this reason, they can be very hard to observe if they arise late in the design cycle. This class of bugs can be associated to situation where particular RTL coding styles lead to different semantics pre- and post-synthesis, but also to specific real-world conditions, such as interferences, that are only observable on-chip. Due to their nature, not only they are difficult to reproduce, but they are also difficult to analyze, because the necessary information needs to be collected from the chip when the design runs at full speed. The work described in this thesis also demonstrate how Discrepancy Analysis can be successfully brought to on-chip debugging, with minimal overhead.

To summarize, the goal of this work is to showcase a methodology for automated bug detection in hardware generated with High-Level Synthesis, that is able to handle the following classes of bugs:

- bugs introduced by HLS tools;
- bugs introduced in system integration, due to wrong interconnections, to faulty modules from external IP libraries, or to bad wrong use of the reused components;
- bugs coming form intrinsic differences between hardware and software semantics in case of *undefined behavior* or *unspecified behavior* in C;
- post-synthesis bugs of different kinds, ranging from electrical problems on chip, to bad RTL that leads to mismatches between pre- and post-synthesis behaviors.

These classes of bugs will be discussed in detail in the results of the thesis, in Chapter 11.

Summary

This chapter provided a high-level overview of the ideas behind the work described in the thesis. Section 4.1 introduced the problem, Section 4.2 the key insights on how *Discrepancy Analysis* tries to solve it, and Section 4.3 clearly stated the goals and the intended features of the approach. Finally, Section 4.4 focused on the classes of bugs that the proposed approach has been designed to solve. Chapter 5 will now define the concept of equivalence between hardware and software execution, which is at the basis of all the rest of the work. This Hardware/Software Equivalence is the conceptual framework that allows to decide if two executions are equivalent and to identify mismatches. It is the core of two different flows for automated bug detection, described in Chapter 6. The details of how the high-level concepts introduced in this chapter are used in each flow are expanded in Chapter 8, 7, and 9.

5 Equivalence Between Hardware and Software Execution

The approach described in this thesis is based on the idea of automated comparison between hardware and software executions, to find mismatches without user interaction. In order to do this, it must be possible to define when hardware and software executions are equivalent on the same inputs. This chapter introduces this notion of equivalence, at two levels: *Control Flow Level*, in Section 5.1, and *Operation Level*, in Section 5.2. Section 5.3 summarizes the two levels in a single definition of equivalence that encompasses both of them. In addition, Section 5.4 describes a generic workflow for Discrepancy Analysis, showing how these concepts can be used in practice.

Part of the material composing this chapter was originally published in international peer-reviewed conference proceedings [22]: P. Fezzardi, M. Castellana, and F. Ferrandi. Trace-based Automated Logical Debugging for High-Level Synthesis Generated Circuits. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 251–258, Oct 2015.

5.1 Control Flow Level

The first important way to compare hardware and software executions is looking at control flow. For software, control flow is represented statically by the Control Flow Graph (CFG), built by the compiler front-end and used for all the front-end optimizations. For hardware, the same information is represented by the Finite State Machine (FSM) generated during the HLS process, along with a DataPath. HLS can also perform several non-trivial modification and optimization on the IR to generate FSM and DataPath, like explained in Section 2.1. In order to define *Control Flow Traces* and *Control Flow Equivalence*, what is important is that during HLS every Basic Block in the CFG is mapped onto a chain of states in the FSM, with jumps and branches only at the end of the last state in the chain. This is not strictly true if the FSM implementation used in HLS tool is based on guard conditions [18], but the definitions can be refined to support also this case. An example of this mapping is shown for example in Figure 5.1.

The equivalence at control flow level is defined per-function. Consider a function `f` described in a high-level language such as `C`, its Control Flow Graph after front-end optimizations, and the Finite State Machine generated from it with HLS. With the appropriate conventions, the two graphs (CFG and FSM) can accept the same inputs. On a given input, the CFG represents the control flow of the execution of the software and the FSM the execution of the generated hardware. 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. Intuitively, hardware and software execution are equivalent if the Basic Blocks traversed during execution by software match the state traversed by the Finite State Machine on the same given input. From this observation are ensued the definitions of *Software Control Flow Traces* and *Hardware Control Flow Traces*. *Control Flow Equivalence* is then defined operationally starting from the concept of traces.

Definition 1. Given the Control Flow Graph of a high-level function `f`, the *Software Control Flow Trace* (SCFT) of `f` on a given input I is the ordered sequence of Basic Blocks traversed on the CFG during the execution of `f`.

Definition 2. Given the Finite State Machine generated with High-Level Synthesis from the same function `f` of Definition 1, the *Hardware Control Flow Trace* (HCFT) of `f` on the input I is the ordered sequence of states traversed by the Finite State Machine during the execution.

Definition 3. Software Control Flow Traces and Hardware Control Flow Traces 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 5.1 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 5.1 the Hardware Control Flow Trace is <**S0_0**, **S0_1**, **S0_2**, **S0_0**, **S0_1**, **S0_2**, **S1_0**, **S1_1**, **S3**, **S3**>.

Control Flow Traces are the elementary components used to define equivalence between the execution of a function `f` in software and the hardware component generated with HLS to implement `f` on an FPGA.

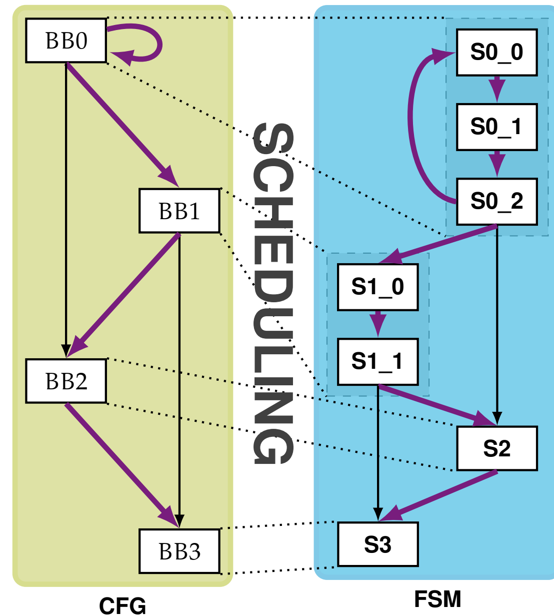


Figure 5.1: Relationship between Control Flow Graph and Finite State Machine. Purple thick arrows show the HW and SW executions.

Definition 4 (Equivalence of Control Flow Traces)**.** Consider the Control Flow Graph and the Finite State Machine generated during High-Level Synthesis from a high-level function `f`. Let be fixed an input I for both the CFG and the FSM. Let then be $S_{cf}(I) = <BB_0, BB_{k1}, BB_{k2}, \ldots, BB_{K(I)}>$ and $H_{cf}(I) = <S_0, S_{j1}, S_{j2}, \ldots, S_{J(I)}>$ the Software and Hardware Control Flow Traces of `f` on input I. $S_{cf}(I)$ is *equivalent* to $H_{cf}(I)$ if $H_{cf}(I)$ can be produced from $S_{cf}(I)$ substituting (BB_k) with the states associated with it through scheduling. The control flow equivalence between the *Control Flow Traces* is represented with the notation $S_{cf}(I) \equiv_{cf} H_{cf}(I)$.

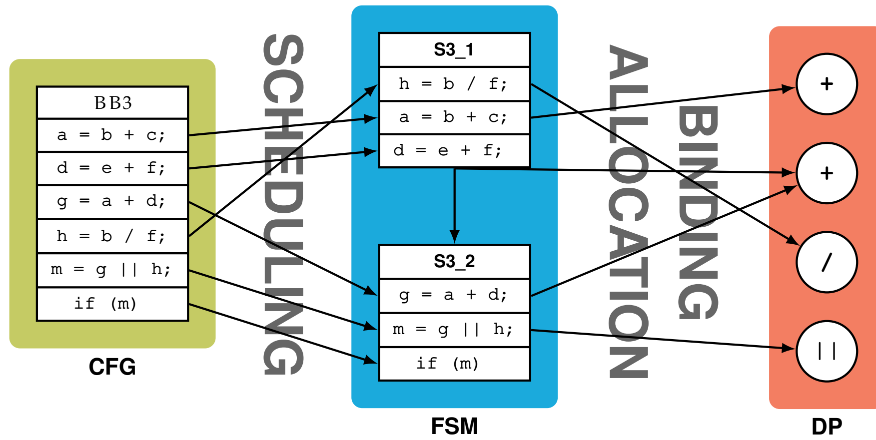


Figure 5.2: Scheduling, allocation, and binding mappings. Scheduling assigns every operation in a Basic Blocks to a state in the Finite State Machine; allocation selects the kind of hardware component in the DataPath used to implement the operation; binding identifies which specific instance of a given component type is used to execute the operation in the DataPath.

Definition 5 (Control Flow Equivalence)**.** Consider a high-level function `f`, its Control Flow Graph and the Finite State Machine generated from it with High-Level Synthesis. The software version of `f` is *Control Flow Equivalent* on a given input I to the hardware version generated with High-Level Synthesis if and only if $S_{cf}(I) \equiv_{cf} H_{cf}(I)$.

This defines *Control Flow Equivalence* between hardware and software execution on a given input I. Notice the dependency on the input, which means that hardware and software can be equivalent on a given input I and not on another input J. This is part of the methodology, which aims at automated bug detection when a bug is present but is not concerned on how to define the input values to ensure that there are no bugs. An interesting but orthogonal direction of research could investigate how to carefully choose a technique to generate the input data sets to increase the coverage of the tests, but it is beyond the scope of this work. Ideally, the approach described here can be coupled with existing methods to increase code coverage, in order to reduce the chances of bugs hidden in deep corners of the code and that are not exercised by the tests.

5.2 Operation Level

Control Flow Level can be a good starting point for automated bug detection. It is already enough to tell if two executions are not equivalent but it cannot pinpoint the root cause. Moreover, if a bug does not alter the control flow it is invisible at Control Flow Level. To overcome these limits it is necessary to refine the granularity up to single operations, considering also HLS information from binding and allocation. Figure 5.2 shows how the list of statements in a BB can be reordered and assigned to operations scheduled in different states of the FSM. The dashed arrows on the right represent how the operations are bounded to allocated components in the DataPath. Note that the mapping of operations on hardware 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 schedule a single operation in multiple states in the FSM, to reduce the number of execution cycles. The key observation is that, when a single operation is duplicated in more than a state, 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, then 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 6. Let O_i be an operation in a Basic Block. The *Software OpTrace* (SOT) of O_i is the list of the results $s_{1,i}, \dots, s_{k(i),i}$ of the operation across all the execution.

Definition 7. 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 the 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 the state $S(O_i)$.

Definition 8. Software OpTraces and Hardware OpTraces together are often designated with the general term *OpTraces* in the following.

Notice that in these definitions the dependence on the input I provided to the function has been omitted, to avoid to weigh down the notation. However, just like for *Control Flow Traces*, *OpTraces* depend on the input provided to the function where the operations belong. Likewise, they also depend on the whole execution of the rest of the application, which also depends on the input. The reason is that the exact result of a single operation could depend on some global state that is changed by previous operations. This is why High-Level Synthesis tools and compilers in general usually rely on Control Data Flow Graphs, which are Control Flow Graphs extended with data dependencies between operations. Control Data Flow Graphs are not used in this thesis because they do not provide additional information that is useful for the methodology. However, one of the groups of bugs that Discrepancy Analysis is able to catch is caused by the wrong reordering of operations due to missing data dependencies. More details are given in Part III.

As stated above, multiple operations can be mapped onto the same component in the DataPath. This represents a challenge in detecting the correct value for the result of a given operation. The reason is that the output signal of the underlying component can represent values related to different operations at different times in hardware execution. However, if a component is shared between multiple operations, they must be scheduled in different states, so it is enough to pick the output value of the component when the FSM is in the correct state to retrieve the correct result for the operation. In order to do this, it is necessary to cross-correlate *OpTraces* with *Control Flow Traces*. This has to be done in two different

ways depending on the nature of the operations: Fixed Latency Operations (FLO) or Variable Latency Operations (VLO).

For FLOs the execution time is fixed and known to the HLS tool, as are the states of the FSM where operations are scheduled and where their execution finished. These data are all that is necessary to untangle different HOT from output signals of components shared among FLOs.

VLOs, instead, are handled with a handshaking mechanism as described in Section 2.1.2. The handshaking signals can be easily used with control flow information to infer the real start and end time of VLOs, as well as to untangle multiple HOTs from the output signals of components shared among VLOs.

Definition 9 (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 $C(O_i)$ be the component in the DataPath that was allocated and bounded in HLS to execute the 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. The operation equivalence between the *OpTraces* is represented with the notation $S_{op}(O_i) \equiv_{op} H_{op}(O_i)$.

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, as will be described in Chapter 8.

Similarly to what happens for Control Flow, *Operation Equivalence* on a given input can be defined in terms of equivalence between *OpTraces*. Here the dependency on the input I is explicitly stated, to make it stand out.

Definition 10 (Operation Equivalence)**.** Consider a high-level function `f`, its Control Flow Graph and the Finite State Machine generated from it with High-Level Synthesis. The software version of `f` is *Operation Equivalent* on a given input I to the hardware version generated with High-Level Synthesis if and only if for every operation O_i in `f` $S_{op}(O_i) \equiv_{op} H_{op}(O_i)$.

5.3 Hardware/Software Equivalence

Using the notion of *Control Flow Equivalence* and *Operation Equivalence* on a single function, it is now possible to define equivalence between hardware and software executions in general terms. First, the equivalence is defined for a single function.

Definition 11 (Hardware/Software Equivalence for a single function)**.** Consider a high-level function `f`, its Control Flow Graph and the Finite State Machine generated from it with High-Level Synthesis. The execution of the software version of `f` on a given input I is *Equivalent* to the execution of the hardware version generated with High-Level Synthesis if and only if for every operation they are both *Control Flow Equivalent* and *Operation Equivalent* on the same input I.

Hardware/Software equivalence between a design generated with High-Level Synthesis and the software application used as specification can then be defined composing equivalence on the single functions.

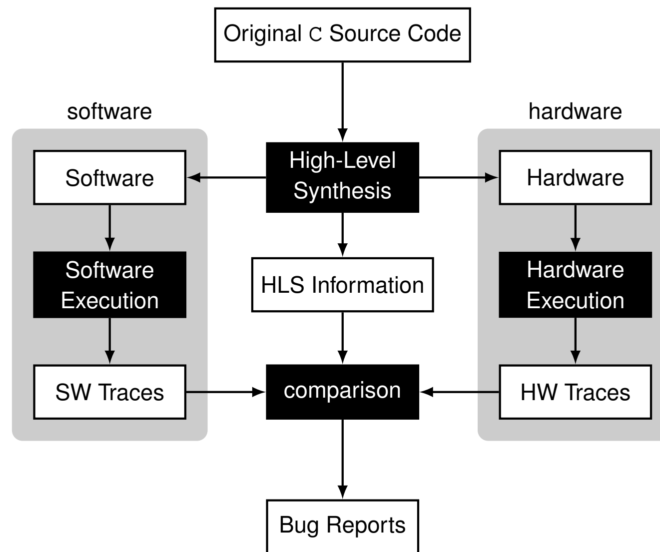


Figure 5.3: Outline of a generic Discrepancy Analysis debug workflow. White boxes represent data, while black boxes represent steps of the workflow.

Definition 12 (Hardware/Software Equivalence for an Application)**.** Consider a program written in a high-level language, used as starting point for High-Level Synthesis. Let F = { `f1`, ..., `fn` } be the set of functions composing the application. The execution the program is *Equivalent* to the execution of design generated from it with High-Level Synthesis if and only if ∀ `f` ∈ F, the software and hardware version of `f` are *Control Flow Equivalent* and *Operation Equivalent*.

5.4 A Generic Workflow for Discrepancy Analysis

The previous sections set up the operational definitions necessary to define Discrepancy Analysis. These definitions allow to clearly identify two complementary ways to analyze the behavior of hardware and software executions, to tell if they are equivalent or not. They have the big advantage of being agnostic about the actual encoding of the traces, and about how these traces are collected in practice. This makes them suitable to be used with traces collected in different ways, and allows to describe a generic design workflow for Discrepancy Analysis.

Figure 5.3 represents a generic workflow for Discrepancy Analysis, that can be created composing the notions introduced up to this point. In the figure, boxes with white background represent data, while boxes with black background represent steps of the workflow that manipulate the data. The flow starts from the original high-level C code that is the input to the High-Level Synthesis tool. This code is compiled, producing two outputs: hardware and software. Both are then executed to generate the respective traces, described in the previous sections. Finally, at the bottom, Software Traces and Hardware Traces are compared to generate bug reports useful for users.

This high-level outline of the process can then be customized and adapted for different scenarios. For example, it is possible to generate hardware designs at the RTL level and execute them in a simulator to extract the traces, or to go directly on chip and execute the design in a real environment. Another choice concerns when to execute the actual comparison. One can decide to compare the traces on

the fly while they are still being generated, or to wait until the end of execution to compare them offline. Depending on these choices it is possible to create different debug flows based on Discrepancy Analysis, some of which are more suitable for use in certain environments or to find specific classes of bugs. Chapter 6 will describe how this

Summary

This chapter laid the foundation of the *Discrepancy Analysis*, defining the fundamental concept of equivalence between hardware and software execution. *Discrepancy Analysis* exploits this model to compare the executions at two levels, Control Flow and Operation, as described in Sections 5.1 and 5.2 respectively. These two levels are then stitched together to provide a global concept of equivalence, described in Section 5.3. These concepts can be used to outline a generic workflow for automated bug detection with *Discrepancy Analysis*, like explained in Section 5.4. Chapter 6 describes two different flows, showing that Discrepancy Analysis can actually be used in different scenarios, with workflows using traces with a variety of encodings and collected in various ways. At the same time, the definitions provided in this chapter are quite abstract and it may be difficult to understand how they can be used without practical examples. The two flows described in Chapter 6 also serve this purpose.

6

Discrepancy Analysis: Two Different Flows

The definitions of equivalence described in Section 5 do not depend on how the execution traces are collected, nor from the fact that their analysis happens online or offline. For this reason, this chapter introduces two separate *Discrepancy Analysis* workflows that employ the same underlying equivalence model to implement two different debug frameworks. In this chapter, the two workflows are described in general terms at a high-level, for a better understanding of the two modes of operation. Section 6.1 briefly introduces the High-Level Synthesis tool that was used to build a reference implementation of both the debug flows, as a proof-of-concept of the approach and to measure its effectiveness. The details of both of them are discussed later in this thesis.

Section 6.2 describes a workflow for offline bug detection, which collects the hardware traces with simulation. Using simulation, it is trivial to achieve complete observability of all the signals throughout the entire hardware execution. This allows showcasing the full potential of the *Discrepancy Analysis* without worrying about resource constraints on FPGA. Performing offline analysis of the traces also avoids the necessity to guarantee real-time comparison of the traces. Chapters 7 and 8 describe the details of this workflow.

Section 6.3, instead, shows a flow for on-chip online *Discrepancy Analysis*. This allows to demonstrate the flexibility of the *Discrepancy Analysis*, proving that it can be used online on FPGA with minimal impact in terms of area and frequency. In addition, it shows and that it actually has some advantages in terms of memory footprint compared to other state-of-the-art techniques. The techniques underlying this online on-chip workflow will be described in detail in Chapter 9.

6.1 Reference Implementation

Both the workflows have been implemented in *PandA*, an open source framework for Hardware/Software codesign, developed at Politecnico di Milano. The *PandA* framework includes a High-Level Synthesis compiler, called BAMBU [60], based on GCC [79]. BAMBU has an advanced approach to memory allocation [67] and it implements fairly complex state-of-the-art front-end optimizations [49]. BAMBU accepts C as input language, with support for a subset of the OpenMP specification [83]. It is capable of generating designs in Verilog and VHDL, targeting a wide range of modern FPGA devices from multiple vendors: Xilinx, Altera (now Intel FPGA) and Lattice Semiconductors.

The support for advanced optimization is important for this work because it allows showing that the *Discrepancy Analysis* algorithm can handle all the possible optimizations performed during HLS, without imposing unrealistic restrictions.

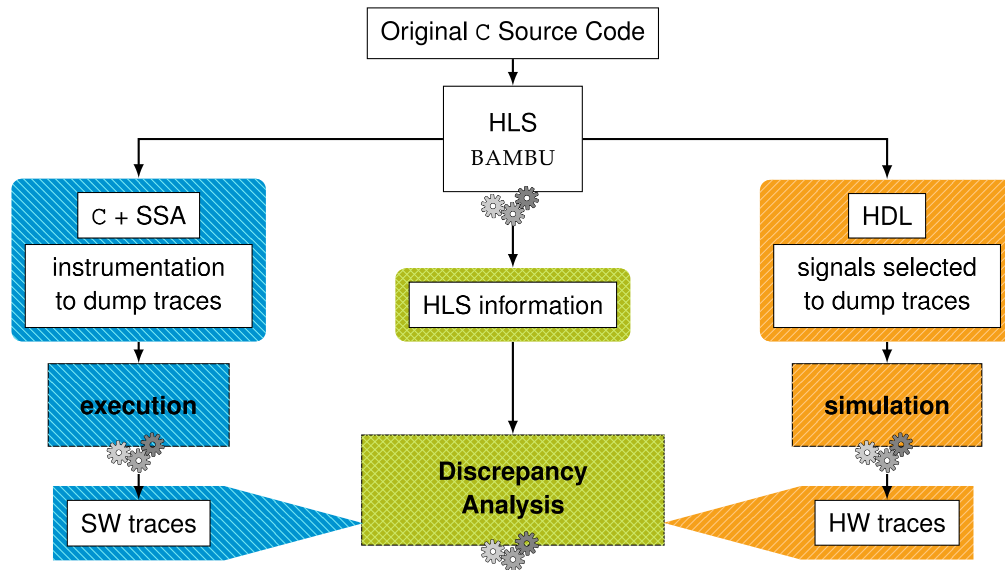


Figure 6.1: Outline of the offline *Discrepancy Analysis* debug flow based on simulation.

At the same time, the advanced memory allocation engine allows to stress-test the extension of *Discrepancy Analysis* to pointers and addresses, described in Section 8. The OpenMP support is another important component because it allows to validate the approach in case of debugging of parallel hardware designs generated with HLS from multi-threaded programs.

These were the main reasons that drove the choice of BAMBU among other open source HLS tools. In addition, BAMBU features an extensive regression test suite, based on the GCC `C-torture` tests [25] and on the *CHStone* benchmarks [33]. The `C-torture` tests [25] are a broad set of self-contained C programs, specifically designed to "torture" compilers, to ensure that they are compliant with all the most obscure parts of the C standard. The *CHStone* benchmarks [33], instead, are composed of 12 examples representing different typical use cases of HLS. Using these two sets of benchmarks allowed to explore all the capabilities of the methodology proposed in the thesis. It also allowed spotting most of the early stage limitations, allowing various subsequent improvements.

6.2 Simulation-Based Offline Discrepancy Analysis

The debug flow for offline *Discrepancy Analysis* based on simulation is depicted in Figure 6.1. The original C source code is initially processed by the BAMBU HLS compiler. During the HLS process, the compiler gathers information about all the optimizations and transformations it performs. Then, it originates two separate flows for the collection of hardware and software traces.

The portion involving software is in blue on the left. BAMBU prints back the IR after front-end optimization, along with instrumentations for the generation of the software traces. The IR is printed back in C, using a code generator that is designed to maintain the closest resemblance to the original source code. At the same time, the variables are versioned in *Static Single Assignment* form (SSA, see Section 2.1.1). Phi operations, that have no direct representation in C, are converted into assignments, as explained by Briggs et al. [10]. Using SSA allows

constructing a bijective relationship between SSA variables and assignments because with SSA every variable is only assigned in a single statement. This greatly simplifies the analysis of the software traces, as will be explained in Chapter 7. The C code instrumented for trace generation is then compiled and executed, and the traces are written to a file in a machine-readable format. More about their content is described in Chapter 7. The generation of software traces concludes the software flow.

The portion involving hardware is in orange on the right. BAMBU generates an HDL design for the input program, along with a list of the signals selected in the design for the generation of the traces. Notice that the generated HDL is exactly the same that would be generated by BAMBU without the *Discrepancy Analysis*. This design is used for RTL simulation with cycle accuracy. The experiments shown in Part III have been executed with ModelSim SE-64 version 10.5 from Mentor Graphics, but in general the approach does not depend on the simulator used for the generation of the traces. The input values used for the simulation are the same used for the software counterpart. The traces are extracted during simulation, printing them in some waveform format, like Value Change Dump (VCD). This operation completes the hardware flow.

Finally, the software and the hardware traces are fed into the *Discrepancy Analysis* automated bug detection algorithm (in green in Figure 6.1). The bug detection is executed offline, but it has access to all the information coming from HLS. In this way, it can manage all the complexity of bug detection, without requiring any intervention from users. If a mismatch is found, all the HLS information are provided to the designer, along with other data useful to reconstruct what went wrong. Some of these data are for example: the first mismatch between hardware and software executions, the name of the involved signal and its hierarchical path, start and end time of the failing operation in hardware, the corresponding variable and operation in the high-level source code, the state of the FSM when the mismatch happened, and the value of the wrong signal in hardware compared to the expected value from C. This greatly simplifies the job of designers looking to fix their designs.

6.3 On-Chip Online Discrepancy Analysis

For on-chip online *Discrepancy Analysis*, the debug flow shown in Figure 6.1 has been modified to run on-chip. In particular, the details of the new workflow are explained in Chapter 9 and the proposed methodology only focus on *Control Flow Level*. This is already enough to show the feasibility of on-chip *Discrepancy Analysis*. Moreover this already shows relevant advantages in terms of memory footprint, area overhead and frequency implications, compared to other state-of-the-art approaches for on-chip debugging. Another reason is that the technique used to achieve these improvements is intrinsically well suited for *Control Flow*, but not to handle *Discrepancy Analysis* the *Operation Level*. The reasons are explained in detail in Chapter 9.

The online on-chip debug flow is depicted in Figure 6.2. The portion on top with the purple background runs on the host computer, while the portion below with the orange background executes directly on the FPGA. The software traces are generated in the same way described in Section 6.2, but the High-Level Syn-

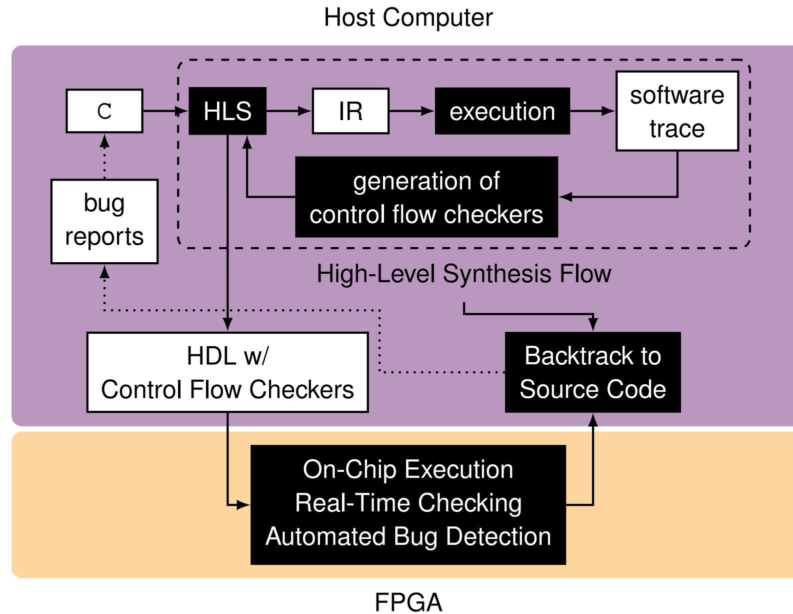


Figure 6.2: Modified *Discrepancy Analysis* flow for online on-chip debugging.

thesis flow, enclosed in the dashed box, has been extended to use the software traces to generate dedicated components, called *Control Flow Checkers*, that perform the control flow checks on-chip.

A customized instance of *Control Flow Checker* is integrated alongside the Finite State Machine of every functional module. This enables a fine-grained customization of the checkers, to use the smallest number of bits necessary for every function. The optimal number of bits and the dimension of the necessary memory depend also on the software reference trace. Given that all these factors can be evaluated on the IR, before the generation of the checkers, it is possible to explore the optimal values of the parameters for every checker before synthesis, as explained in Chapter 9. *Control Flow Checkers* contain memories that are initialized with the expected control flow trace computed starting from software execution (see Chapter 9). This is one of the differences between previous works and the methodology proposed here: the debugging logic is not used to memorize information on the hardware execution, but to compare it in real-time with a golden reference. Others have adopted this strategy [13] [94] relying on simulation instead of debugging on-chip. In this work, the debugging components are integrated into the design by the HLS engine and then synthesized and executed on FPGA. As soon as a checker detects a mismatch between the expected execution and the real behavior, it notifies is to the host. Only little information needs to be extracted by the chip and exposed outside: a unique identifier of the checker instance (that is uniquely determined during HLS and embedded in every checker), and the offset in the trace execution where the mismatch happens.

With this approach, the quality of debugging is at least as good as with offline *Discrepancy Analysis* based on simulation. This means that all the control flow bugs detected with simulation are also visible with the hardware checkers, while with on-chip debugging it is potentially possible to detect post-synthesis bugs and mismatches that come from problems on system integration. The methodology proposed here allows to reduce the memory necessary for the traces while

providing two advantages:

1. improving visibility of bugs that only arise on-chip;
2. automatically backtracking bugs to the original source code.

The first point is an advantage compared to approaches that only rely on simulation [13] [94]. The second is not possible with other approaches that focus only on providing architectural support for the collection of the traces [26], leaving the burden of their manual analysis entirely to users.

Summary

This chapter started by briefly introducing, in Section 6.1, the High-Level Synthesis tool that has been used as a reference implementation for the work described in the thesis: BAMBU. Then two different flows implemented in BAMBU and based on *Discrepancy Analysis* have been outlined. The first, *Simulation-Based Offline Discrepancy Analysis*, was explained in Section 6.2. The second, *On-Chip Online Discrepancy Analysis* was outlined in Section 6.3. The possibility to use the hardware/software equivalence shown in Chapter 5 to perform *Discrepancy Analysis* in such different debug flows shows the flexibility of the approach.

The remainder of Part II is divided into three chapters. Chapters 7 and 8 are about *Simulation-Based Offline Discrepancy Analysis*. Among other details, they also explain how it can be used to debug hardware designs obtained from multithreaded program specifications (Chapter 7), as well as pointers and memory accesses (Chapter 8). Chapter 9, instead completely describes the *On-Chip Online Discrepancy Analysis*, based on a software profiling technique called *Efficient Path Profiling* [7], that has been adapted to Finite State Machines.

7

Simulation-Based Offline Discrepancy Analysis

The Simulation-Based Offline Discrepancy Analysis debug flow introduced in Section 6.2 is described in general terms. This is useful to have a big picture of what is going on, but it does not give any practical technical detail about the underlying operations involved in the process. Moreover, the theoretical definitions in Chapter 5 do not explain how the traces can be extracted 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. However, in order to understand how to automatically detect bugs, it is necessary to provide a description the format and the encoding of the traces, and an algorithm to compare them.

The aim of this chapter is to fill these gaps, for what concerns the Simulation-Based Discrepancy Analysis debug flow. Section 7.1 briefly explains how to generate and collect execution traces from hardware and software execution, along with a method for automatic identification of the necessary signals in the generated designs. The format and the encoding of the traces are also described. Section 7.2 describes an algorithm for the comparison of the traces to check for equivalence between hardware and software executions. The comparison can be performed separately on control flow and data level. Separating the two levels makes the algorithm easier to understand and faster to execute. However, the two levels are based on the same method for fast pattern-matching, leveraging Finite State Automaton (FSA). Finally, Section 7.3 extends the pattern-matching algorithm to hardware generated with HLS from multi-threaded programs.

Part of the material composing this chapter was originally published in international peer-reviewed conference proceedings [22]: P. Fezzardi, M. Castellana, and F. Ferrandi. Trace-based Automated Logical Debugging for High-Level Synthesis Generated Circuits. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 251–258, Oct 2015.

7.1 Generating and Collecting Execution Traces

Like explained in Section 6.2, hardware and software traces are generated and collected separately. Hardware traces are obtained from the simulation of the design generated with High-Level Synthesis, while software traces are printed from instrumentations during the execution of the high-level source code. These two flows are discussed here in detail.

7.1.1 Software Traces

For the generation of the Software Traces, the high-level source code must be instrumented to print the necessary information during the execution. The instrumentations are added directly in the IR to have a finer granularity and gain 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 [20]), and it prints it back in `C` splitting SSA's ϕ operations as described in [10]. In this way, all the operations that are not control flow instructions can be printed as assignments. In the IR in SSA form, every variable is assigned only once, in the same way as every printed statement assigns only one variable. In this way, to generate the *Software OpTraces* it is enough to print the value of every variable (which has a unique identifier) after its assignment in software. To generate the *Software Control Flow Traces* a print instruction is placed at the beginning of every BB, to print the identifier of the BB itself each time its execution stars. In practice, the traces are written to a file, that is then parsed and fed into the bug detection engine for comparison with the hardware traces. The details of the comparison are explained in Section 7.2, while here the format of each trace is discussed.

A *Software Control Flow Trace* is generated for every function in the program. For every function, the associated *Software Control Flow Trace* is a list containing all the identifiers of the Basic Blocks traversed during the entire execution.

A *Software OpTrace* instead is associated with every operation. Given that the program is restructured in SSA, this is equivalent to associating a *Software OpTrace* to every SSA variable identifier. Every *Software OpTrace* is a list of all the values assigned dynamically to that SSA variable during the execution of the program. Thanks to the SSA form, this is equivalent to the list of results of all the executions of that operation at runtime.

Notice that neither *Software Control Flow Traces* nor *Software OpTraces* contain timing information. *Hardware Traces*, instead, contain timestamps of signals variation, as explained later. This is an intrinsic difference due to the two execution models. It is also an issue that must be dealt with when reconstructing the relationships between hardware and software traces.

7.1.2 Hardware Traces

The first fundamental operation necessary to generate and collect hardware traces is to identify in the design the relevant signals. Indeed, not all the signals in the design have meaningful counterparts in software. Tracing all the signals during all the simulation of the circuit have significant drawbacks. One of the main problems is that modern simulators perform optimizations on their own IR before the actual simulation. If some signals have to be traced during simulation, this limits the number and the effectiveness of these optimizations, slowing down the simulation process. Another issue is that simulators usually write the traced signals to files in waveform formats. Increasing the number of analyzed signals means increasing I/O operations during simulation and enlarging the size of the trace files. This results in even bigger slowdowns, and it can even lead to generate trace files that are unmanageable because they are too big.

For all these reasons, it is a common practice to select only a restricted number of signals to trace during simulation and debugging. When High-Level Synthesis is involved it is clearly unpractical to ask users to select those signals manually. The approach described here performs the selection entirely without user interaction because all the necessary information is already available in the HLS engine.

First of all, 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 others signal variations. This signal is named `clock` in the following.

Similarly to what happens with software, a *Hardware Control Flow Trace* must be generated for every high-level function. According to the definitions of Chapter 5, *Hardware Control Flow Traces* are lists of states traversed by the FSM during execution. Hence, the signal used to produce them is basically the signal representing the state 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; another one, 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 may be necessary to check for `start` and `done` to have the full information on the execution. Summarizing, the necessary signals to produce all the *Hardware Control Flow Traces* 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 Chapter 5, *Hardware OpTraces* are composed of the values of the output signals of the hardware 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 vary 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 chapter relies on simulation because it is the easiest way to provide full observability on the necessary signals and registers without altering the design. However, the approach described in this work could be applied to traces directly collected on-chip, as long as it is possible to provide observability on the necessary signals, as demonstrated in Chapter 9. With simulation, the design is executed with the same input as the `C` program and the signal variations are dumped in Value Change Dump format (VCD [38]). 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 I/O time and VCD size, with obvious benefits.

7.2 Comparing Execution Traces with Finite State Automata

This section describes how to compare the traces to check for equivalence. The comparison can be performed separately on *Control Flow Level* and *Operation*

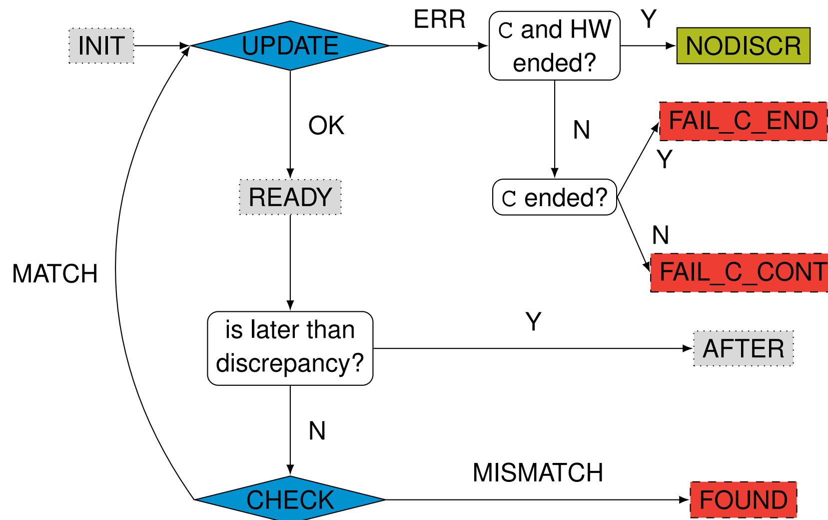


Figure 7.1: Finite State Automaton for the comparison of the traces. Notice that strictly speaking this is not an FSA because it is composed by a mixture of state nodes (with sharp edges) and algorithmic nodes (with rounded edges), resembling a flow diagram. This shows the algorithmic structure of the comparison more clearly. The classical FSA structure can be recovered substituting the nodes with rounded nodes with simple edges with the proper conditions.

Level. Separating the two levels makes the algorithm easier to understand and faster to execute. This section discusses a method for fast pattern-matching of hardware and software traces, based on Finite State Automaton (FSA). Please bear in mind that this terminology is purposely used to avoid confusion with the Finite State Machine of the hardware controller. Notice that the word *status* is always referred to the FSA in the following, while the word *state* always refers to the FSM. Using this kind of automata has two advantages:

1. it makes possible to define a unified algorithm for *Discrepancy Analysis* of both *Control Flow Traces* and *OpTraces*;

2. using stateful checkers can be easily extended to multi-threaded programs later in Section 7.3.

7.2.1 Finite State Automaton for the Comparison of the Traces

Despite the fact that control flow and data traces are fundamentally different in the meaning and in the format, the basic algorithm for their comparison can be based upon a Finite State Automaton with the same structure, depicted in Figure 7.1. Please notice that the automaton depicted in the figure is not strictly speaking a Finite State Automaton, since it mixes states (nodes with sharp edges) and algorithmic nodes (with rounded edges). This is to make the algorithmic flow of the comparison easier to follow, without clobbering the edges of the graph with verbose conditions. The reader can recover a classical FSA substituting the algorithmic nodes with plain edges annotated with the proper conditions. However, the following discussion refers to the graph portrayed in Figure 7.1 as is.

An FSA of this kind works on a pair of associated traces, one for hardware and one for software. The possible statuses of the FSA are represented by rectangular nodes: INIT, READY, NODISCR, FAIL_C_CONT, FAIL_C_END, 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 the last group, FOUND is when an actual mismatch between hardware and software 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 hardware 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 (for details see Section 7.2.2).

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 is 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. If all the data necessary for the evaluation of the following mismatch are available, UPDATE returns 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 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 is executed. CHECK operates on the two next ready values identified by UPDATE on the hardware and software traces. These two values are compared to decide if they actually match. If they do, the cycle restarts with the next entries, otherwise, the FSA enters FOUND and terminates.

7.2.2 Algorithms for Comparison of the Traces

The two algorithms to compare *Control Flow Traces* and *OpTraces* are very similar and share the same structure represented by the FSA described in Section 7.2.1. The only difference between the two algorithms is represented by different operations executed by the UPDATE and CHECK functions in the FSA. Here these operations are described in detail for both the cases.

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 7.2 on traces referred to Figure 5.1. From the figure, it is straightforward to understand how the CFTs can be compared. In this case, the UPDATE and CHECK functions can be unified in a single one, that operates

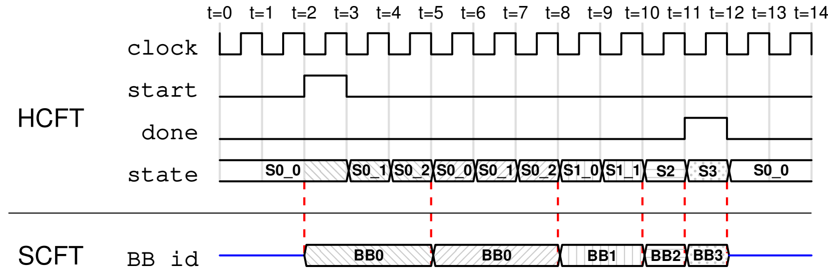


Figure 7.2: Relationship between Control Flow Traces. The HCFT is represented by the first four signals, while the SCFT is the list of Basic Block identifiers. The traces are referred to the CFG and FSM shown in Figure 5.1. The red dashed lines between `state` and `BB id` represent the scheduling relationship between states and basic blocks.

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**

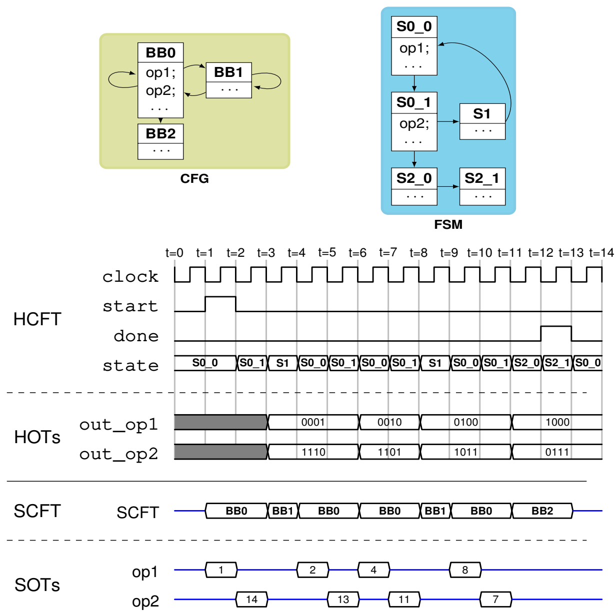


Figure 7.3: Visualization of Hardware and Software Traces, with the CFG and FSM, used to generate them.

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 with that basic block. This mapping is depicted with red dashed arrow in the figure. Finally, it checks that the `state` signal in the hardware traces are coherent with the identifiers computed from scheduling. The `clock`, `start`, `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 7.3 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 hardware and software traces, and it fills a map of discrepancy reports for every operation. The main loop works on a single variable at a time, selecting hardware and software traces and passing it to the FSA described in Section 7.2.1. The result of the ex-

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;
```

ecution 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 could be treated FLOs but, unless there is 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 this kind of operations.

The UPDATE function is described in Algorithm 2. It starts flowing 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 hardware execution (line 2). Lines 3 to 5 perform some sanity checks on the new start time and on the SOT. If the SOT is empty there is nothing to compare the hardware 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 hardware, so it is marked as an error. From line 6 to 13, the end_time in hardware of the newly started operation is computed. This is necessary to compare the output signal with the value in `C` only after the operation is complete. For FLOs, execution time is fixed and known, 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 before the end of the simulation, UPDATE returns ERR, otherwise OK.

As explained in Section 7.2.1, CHECK accepts as input the two next values identified by UPDATE on the hardware and software traces. The CHECK operates in different ways depending on the type of the variables associated with the *OpTrace*.

For integer variables it is a simple bitwise comparison, but for other types it requires more complex operations. A notable example are floating point variables. Floating point data they have two separate representations for zero, with different values for the sign bit. Another subtlety is that the IEEE754 standard allows certain classes of mathematical functions to return results that are wrong up to 1 *Unit in Last Place* (ULP). This means that it is not necessarily a bug if the results of these operations differ in software execution and in hardware. The implementation described in this work allows specifying the max error allowed in terms of ULPs for floating point operations.

The other main type of variables that cannot be compared directly is represented by addresses and pointers. The reason is that there is an intrinsic difference between the address spaces in hardware and in software. Indeed, during software execution, all the variables allocated in the program stay in the same address space, and if two pointers are equal (i.e. they have the same underlying bitwise representation) they forcibly point to the same object allocated in memory. In hardware, this is not true. As explained in Section 2.2, HLS tools perform memory optimizations, including partitioning and restructuring. Hence, high-level variables and objects often end up to be allocated in physically separate memory modules on FPGA. For these modules, the natural counterpart of pointers are the signals used for addressing them. If two different variables reside on two separate memory modules, they can, in theory, be addressed using the same address on the two modules. This means that it is often the case that two "hardware pointers", i.e. two addressing signals used to access different memories on FPGA, have actually the same underlying bitwise representation, but they actually point to two different objects. The pointed objects can even have different types. This situation makes Discrepancy Analysis of pointers more complicated. Chapter 8 describes the additional information and data structures that must be extracted from the HLS process in order to be able to debug pointer operations.

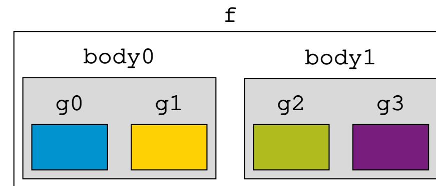
The algorithm on its own may not be enough to understand how the checker FSA works in practice. A simple example can be demonstrated with the traces sketched in Figure 7.3. Consider op1. It is in **BB0** and it is scheduled in **S0_0**, so **S0_0** is a starting state for op1. Assume it is an FLO with execution time of 2 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 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 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 3 times to check all the 4 assignment. The fourth time the UPDATE function is called, it returns ERR, since the SOT is empty and there are no new 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.

7.3 Debugging Circuits Generated from Multithreaded Programs

Discrepancy Analysis as described in Section 7.2 does not support multi-threading, actually not even procedure cloning [84]. The reason is that it silently makes the

```
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;
}
```

(a) Example of a C function containing using an OpenMP parallel for loop to speedup computation.



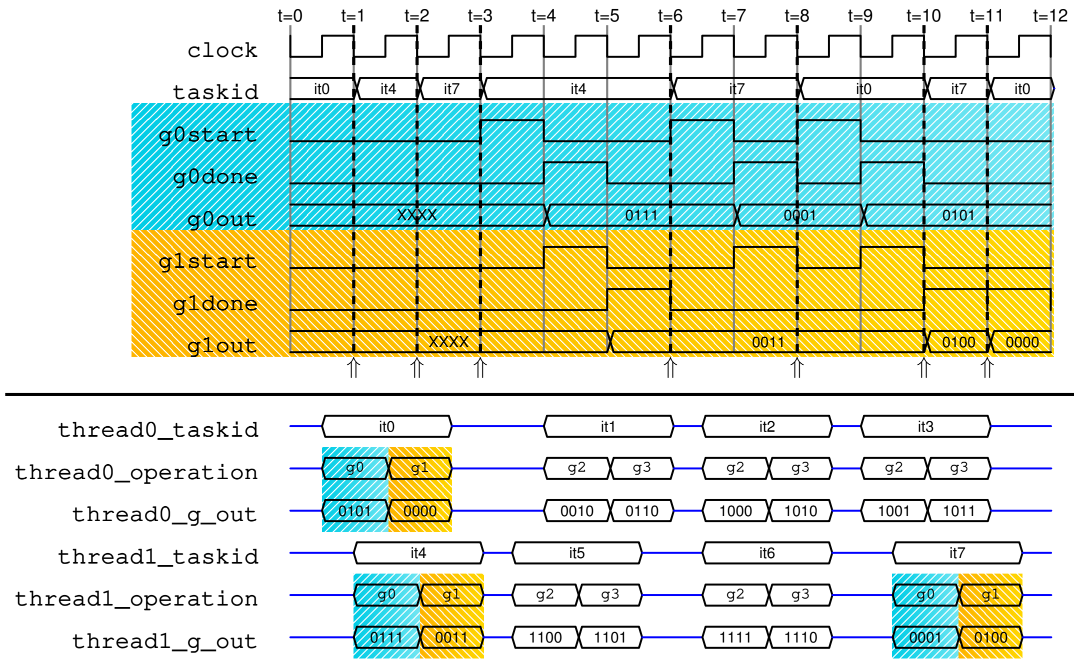
(b) Structural layout of the parallel hardware architecture generated starting from (a).

Figure 7.4: A snippet of C code using OpenMP for parallelization and one of the possible parallel architectures that can be generated from it with High-Level Synthesis.

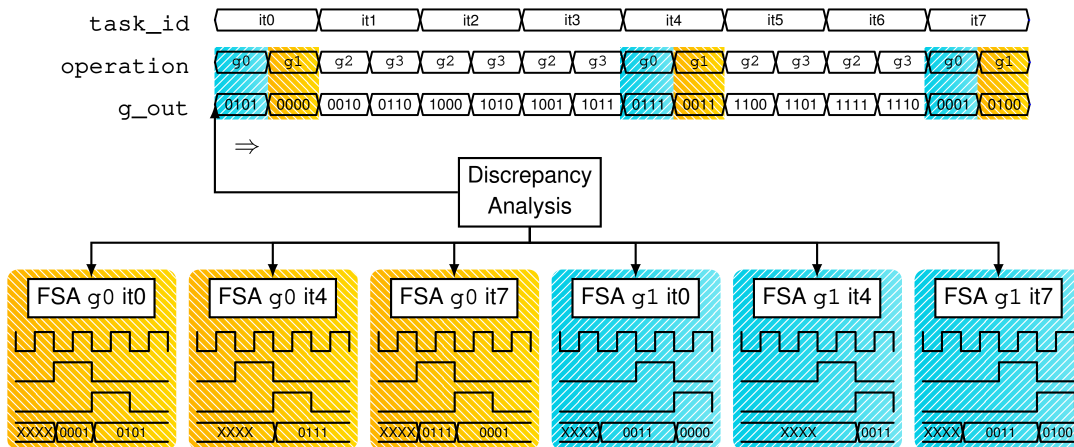
assumption that there is a one-to-one mapping between software and hardware traces. This is equivalent to the assumption that there is always only one hardware accelerator for every high-level function and that every accelerator only executes one task to its completion before starting a new one. But this is not what happens when HLS starts from parallel programming directives.

Consider for example the code in Figure 7.4(a). The function `f` contains an OpenMP for loop with memory accesses and multiple calls to `g`, without data dependencies. Without support for multi-threading, a typical HLS tool would generate a component for `f` and a component 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, which is suboptimal for performance. If the HLS tool supports 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 [17] and BAMBU [16]. Otherwise, a parallel architecture could be generated for the loop body, allowing simultaneous executions of different threads at the same time, using 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 [30] based on ROCCC [86] and by Tan et al. [82]. Theoretically, the two techniques could be used together: it could be possible to physically duplicate the loop body and to enable context switching on all the duplicate copies, depending on some design space exploration trade-offs. At the moment of this writing, no known technique adopts this combined approach, but the rest of this section is going to assume that is possible, in order to show the flexibility of the proposed methodology for automated bug detection. This scenario should settle the more general assumptions as possible: physical and hardware-thread parallelism with dynamic scheduling. In this way, if these assumptions are restricted by a particular HLS implementation (like only allowing physical duplication with static scheduling or other combinations) the approach still works.

Suppose that the HLS tool instantiates two physical copies of the loop body, `body0` and `body1`, like in Figure 7.4(b). Suppose then that the memory accesses are recognized as separate memory locations and the tool also instantiates two



(a) Execution traces as they are obtained directly from software and hardware depicted in Figure 7.4.



(b) Preprocessed execution traces ready for *Discrepancy Analysis*.

Figure 7.5: Visualization of the comparison of the traces generated from Figure 7.4.

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 dispatcher components depending on some policy. Finally, suppose that both the copies of the loop body support dynamic context switching, so that 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 iteration 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 memory 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 depicted in Figure 7.5(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 6 lines, grouped into two blocks of 3 with the same background, represent the start, done and the output signal of each instance of g inside body0. It is possible to 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.

The lower part of Figure 7.5(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 *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 *Discrepancy Analysis*. The preprocessing is different for hardware and software. The software traces are merged and filtered according to the sequence of task ids, like shown in Figure 7.5(b). 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 7.5(b), is that there is 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 7.2.2 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 7.2.2. 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 FSA is stateful, it is not a problem if the trace associated with 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.

Summary

This chapter was focused on *Simulation-Based Offline Discrepancy Analysis*. Section 7.1 described how to generate and collect the execution traces for software and for hardware, also explaining how to automatically select in the design the signals necessary to perform automated bug detection. Section 7.2 discussed the algorithm for automated bug detection. The algorithm was described in a generic fashion, based on a Finite State Automaton that is suitable for the comparison of both Control Flow Traces and OpTraces, requiring only a few dedicated customizations to handle the two cases. Finally, Section 7.3 showed how the algorithm can be extended to handle hardware designs generated with HLS from multithreaded program specifications.

Chapter 8, will explain how this baseline implementation of *Discrepancy Analysis* can be extended to perform automated bug detection on pointers and memory accesses. Chapter 8 concludes the discussion of *Simulation-Based Offline Discrepancy Analysis*, while Chapter 9 will later describe *On-Chip Online Discrepancy Analysis*, concluding Part II of the thesis.

8

Debugging Pointers and Memory Accesses

This chapter focuses on the extension of the *Discrepancy Analysis* to support the debugging of operations involving pointers and addresses. As explained in Section 7.2.2, this requires defining additional information and data structures that must be extracted from the High-Level Synthesis process, in order to be able to compare software pointers with signals representing addresses in hardware. To this end, Section 8.1 introduces the *Address Space Translation Scheme* (ASTS), that uses the concept of *Memory Locations* introduced in Section 2.2 to construct a table used to implement the CHECK function for pointer types. Then, Section 8.2 actually describes how the comparison is performed using the ASTS, while Section 8.3 discusses a set of techniques that can be used to avoid false positives.

Part of the material composing this chapter was originally published in international peer-reviewed conference proceedings [23]: P. Fezzardi and F. Ferrandi. Automated Bug Detection for Pointers and Memory Accesses in High-Level Synthesis Compilers. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016.

8.1 Address Space Translation Scheme

The main purpose of this section is to define the *Address Space Translation Scheme* (ASTS). The ASTS is a table that allows comparing *Hardware Memory Locations* with *Software Memory Locations*, despite their different underlying address spaces. Its construction is entirely based on the concepts of *Memory Locations* introduced in Section 2.2. The notation used here is the same.

In the High-Level Synthesis process, memory allocation has to decide for every variable i a *Hardware Memory Location* $\langle M_i, B_i, S_i \rangle$ and to instantiate the necessary memory modules. In software, the same variable i will be allocated at runtime at a certain *Memory Location*, not known at compile time. This *Memory Location* can be represented in a shorter form for software: $\langle CB_i, CS_i \rangle$. Here CB_i is simply the address of the variable and CS_i its size. However, in software, the *Memory Location* where a variable is allocated at runtime, does not depend only from the identifier i. Moreover, different executions of the same program could be mapped at different offsets in the virtual memory by the Operating System. In practice, this means that *Software Memory Locations* are not deterministic and it complicates the problem of their comparison with *Hardware Memory Locations*.

Consider a program where the `main()` calls another function `fun()` multiple times, and `fun()` has a local stack-allocated variable accessed by address. In this case, at every call to `fun()`, the *Software Memory Location* for the same variable will be different. On the other hand, multiple strategies can be adopted from

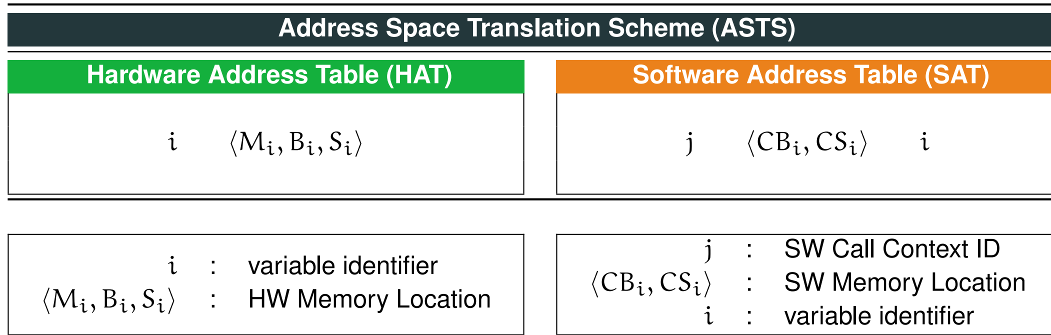


Figure 8.1: Representation of the *Address Space Translation Scheme*

the HLS engine for the synthesis: `fun()` could be inlined; a separate module for `fun()` could be generated; the module used for the implementation of `fun()` could be duplicated and instantiated once for every different call site. Depending on this decision the *Hardware Memory Location* for the local variables of `fun()` may vary. However, this decision must be taken during the HLS allocation step, and it is fixed at the end of the HLS flow itself.

This inherent difference between hardware and software is very much similar to the issues described in Sections 2.3 and 7.3 about the different possible mappings between hardware and software threads. For this very same reason, it can be resolved with a similar mechanism. The key observation is that memory locations are fixed for hardware, but they can vary in software, either between different executions of the same program or between different calls to the same function during a single execution of the program. From this observation, it becomes clear that it is necessary to collect runtime data about the *Software Memory Locations* from software in order to resolve the hardware/software address mapping. For this reason, the same strategy adopted for multi-threaded programs can be used here: enriching software traces with information about memory locations. Another important observation is that *Software Memory Locations* can be actually created and destroyed only when a function is called or when it returns. The reason is that stack memory in software is only allocated and freed on function calls and returns. This also means that the model described here does not consider dynamic memory allocation with `malloc`/`free` functions. This is not a severe limitation because dynamic memory allocation is rarely used in High-Level Synthesis. With this additional assumption, additional memory profiling instrumentations are injected into the `C` code, before it is compiled for the generation of the software traces. In this way, it is possible to dump the software memory mapping at runtime and to extract the data necessary to build the ASTS.

In particular, for every function call, the instrumentations generate and collect the unique context id j. Then, for every memory-mapped variable visible in that scope, they print its identifier i and its *Software Memory Location*, composed of its base address CB_i and size CS_i. To support array partitioning across memory modules, this must be refined. In particular, i must be a unique identifier for an element in the array (or field in a `struct`).

Starting from this information, it is possible to define the *Address Space Translation Scheme*. An outline of its structure is shown in Figure 8.1. The ASTS is subdivided into two tables, which can be implemented efficiently as hash tables.

The first table is called *Software Address Table* (SAT). In the figure, it is on the right, with an orange header, and it contains data on *Software Memory Locations*. One row is in the form $[j, \langle CB_i, CS_i \rangle, i]$. Here j is the primary key and $\langle CB_i, CS_i \rangle$ is the secondary key. They allow, for any given call context, to retrieve efficiently the mapping of software addresses on variables. This means that for every given call context j and for every pointer p in that context, a fast lookup in the SAT is enough to determine the variable i where p points. The second table is the *Hardware Address Table* (HAT). In the figure, it is represented on the left, with a light green header. It is composed of the fields $[i, \langle M_i, B_i, S_i \rangle]$. The HAT is built during the memory allocation step of the HLS process. It maps every variable i to a *Hardware Memory Location* $\langle M_i, B_i, S_i \rangle$.

An important thing for both hardware and software *Memory Locations* is that even if they are expressed in this formal notation, they are easily convertible to bit sequences and back to *Memory Locations*. For software, this is trivial, given that CB_i is actually an address. For hardware, the mechanism strictly depends on the implementation, but it is necessarily computed during HLS for memory allocation and to build the address decoding logic.

Another important consideration is that the construction of these tables happens after HLS, so that all the compiler optimizations have already finished. This is particularly important for memory-to-register and register-to-memory transformation passes that could alter the construction of the ASTS, as well as complex partitioning or restructuring optimizations performed during HLS. Building the tables after HLS allows to treat these optimizations.

8.2 Address Discrepancy Algorithm

The *Address Discrepancy Analysis* algorithm consists of a dedicated implementation of the CHECK function in the FSA described in Section 7.2.1. The structure of the FSA is the same, and also all the other functionalities, but the implementation of CHECK needs to take into account the ASTS to compare the traces. Pointers are variables, so the *Address Discrepancy Analysis* actually extends the comparison of *OpTraces*, while leaving the *Control Flow Traces* unaltered. Also, for non-pointer variables, the Discrepancy Analysis of the *OpTraces* works just like described in Chapter 7. For pointers, a bit-per-bit comparison would obviously lead to a mismatch, even if the synthesized address decoding logic is correct, because of the different address spaces.

The implementation of the CHECK function for FSA-based *Address Discrepancy Analysis* algorithm is depicted in Algorithm 3. In the description in Section 7.2.1, the only inputs to the CHECK function were the two next available entries in hardware and software traces. In this case, to support pointers, Algorithm 3 shows that the inputs must be extended to include the software call context j and the *Address Space Translation Scheme*. In Algorithm 3, `sw_address` represents the next entry in the *Software OpTrace*, i.e. the value of a virtual address assigned to a pointer `p` in the context j. The value of `hw_address` is the next entry in the associated *Hardware OpTrace*, that must be compared with software. This value is actually the bitwise representation of a signal used in hardware to address a memory module. This signal is the 'hardware pointer' associated with `p`.

The function initially performs a lookup in the SAT to compute the variable

Algorithm 3 CHECK Algorithm for Address Discrepancy Analysis.

```
Input: j call context identifier
       sw_address: SW address assigned to a pointer p in j
       hw_address value of the signal associated to p in hardware
       ASTS = (HAT, SAT)
Output: true  if sw_address and hw_address mismatch
        false  othewise
 1: i = search(j, sw_address) in SAT;
 2: if (i is found) then
 3:   ⟨Mi, Bi, Si⟩ = search(i) in HAT;
 4:   if (⟨Mi, Bi, Si⟩ is found) then
 5:     expected_hw_address = decodeHW(⟨Mi, Bi, Si⟩);
 6:     if expected_hw_address ≠ hw_address then
 7:       return  true
 8:     else
 9:       return  false
10:     end if
11:   else
12:     /* i is not allocated in memory in hardware*/
13:     return  true
14:   end if
15: else
16:   /* sw_address is not in range for any variable */
17:   return  false
18: end if
```

i pointed to by the address sw_address (line 1). If the lookup fails, it means that the address is not in range for any variable in software. This means that sw_address does not point to any valid *Software Memory Location*. Hence the Discrepancy Analysis cannot give conclusive results because there is no *Software Memory Location* to compare with hardware (lines 15-18). If the lookup in the SAT succeeds, then the variable i is used as a key for a second lookup, this time in the HAT (line 2-3). If this second lookup fails, it means that there is a variable i in software whose address is taken but that is not mapped to memory in hardware. In this case, the function returns an error (lines 11-14). Instead, if also this second lookup succeeds, the computed *Hardware Memory Location* $\langle M_i, B_i, S_i \rangle$ is converted to an integer with the decodeHW function (line 4-5). The decoded value expected_hw_address represents the expected value of hardware address that would point to the *Hardware Memory Location* equivalent to the *Software Memory Location* pointed to by the software address sw_address. Thus, if expected_hw_address $\neq$ hw_address, a mismatch is detected, otherwise, CHECK returns **false**.

The decodeHW is strictly dependent on the implementation. Hence, it is different for every HLS tool since it uses a lot of HLS information on memory allocation, and on how hardware addresses are actually mapped to hardware.

8.3 Refining Address Discrepancy Analysis

The presented approach has to be refined to avoid false positives. The first class of such false positives happens when the synthesized hardware performs a speculated READ. This is perfectly possible in hardware, but in software it may access

```
extern int something(int *p);
int main() {
   int *p, a[32], b[32], res = 0;
   for (p = a; p < a + 32; p++)
      res += something(p);
   for (p = b; p < a + 32; p++)
      res += something(p);
   return res;
}
```

Figure 8.2: A C program causing a false positive

an invalid address, causing a segmentation fault. Hence READ speculation must be avoided. This is not really a problem since none of the currently available High-Level Synthesis tools actually perform it.

Other problems arise when the points-to set for a given address contains two arrays contiguously allocated in memory by the C code. An example is the code in Figure 8.2, where a and b are contiguous. *After* the last iteration of the first loop, p points to b[0], thus it is in-range for b. The reason is that, *after* the last iteration of the first loop, p is set to &a[32], which causes the loop to end. This assignment is actually performed *before* the second loop and before setting p to b. This means that there is a time when p evaluates to &a[32], which in C overlaps with &b[0] but in hardware it may not. This is not a problem in C, but in hardware a and b could even be mapped onto different memory modules. At this point, if the value of p in software is compared with hardware, it is likely to generate a false positive. A possible solution is to insert poisoned redzones between different memory-allocated areas in C, using the AddressSanitizer (ASAN) memory error detector [77], deployed in both GCC and LLVM. The C code used for *Discrepancy Analysis* is compiled using ASAN. ASAN consists of a compiler instrumentation pass and a run-time library which replaces the malloc function. Using ASAN results in two advantages: it avoids memory bugs in the original high-level code used to generate the traces; it avoids false positives cause by contiguously allocated data. Algorithm 3 cannot really say anything about out-of-range addresses because only in-range addresses are actually used for lookups in the HAT. ASAN is a complementary solution to this problem: mismatches for out-of-range addresses are not reported, but ASAN ensures that there are no dereferences. Another option would be to perform static range checking directly in the HLS tool, which is partially done by most compilers with the correct flags. However, static range checking is not always exhaustive in all the cases. Hence, the instrumented code for trace generation would still need to be compiled with a dynamic range-checking library. ASAN solves all these issues at once and it is guaranteed to improve with time, following the development of mainstream compilers.

Summary

This chapter described the theory behind *Address Discrepancy Analysis*, completing the discussion on *Simulation-Based Offline Discrepancy Analysis*. Section 8.1 initially introduced the *Address Space Translation Scheme* (ASTS), a data structure

constructed during High-Level Synthesis using memory allocation information that is at the core of *Address Discrepancy Analysis*. Section 8.2 then described the algorithm used to compare software pointers and hardware addresses, relying on the ASTS. Finally, Section 8.3 described a few precautions that can be taken to make the results of the analysis more sound.

This chapter is the last that is focused on *Simulation-Based Offline Discrepancy Analysis*. Chapter 9 will now discuss *On-Chip Online Discrepancy Analysis*, to complete the description of the methodology proposed in the thesis and to conclude Part II. Part III will then describe the experimental setup and the results obtained during the evaluation of the different flavors of *Discrepancy Analysis*.

9

On-Chip Online Discrepancy Analysis of Control Flow

This chapter shows how the Discrepancy Analysis described in Chapter 7 can be adapted to on-chip debugging. Section 9.1 outlines the motivation behind this choice. The approach proposed throughout the rest of this chapter aims at providing automated bug detection based on Discrepancy Analysis directly on-chip and during online operation of the circuit. To this end, it exploits the HLS information and the structure of the Finite State Machines to generate and integrate optimized components, called *Control Flow Checkers*, for online on-chip debugging. The generated checkers analyze on-the-fly the execution of the Finite State Machines, automatically halting the circuit when a bug is detected, localizing it and providing data about its cause. The approach proposed here only works at the *Control Flow Level*. To reduce memory usage and save precious resources on FPGA, Software Control Flow Traces are compressed with a technique coming from software profiling. This technique, called *Efficient Path Profiling* (EPP) [7], is described in Section 9.2. Section 9.3 explains how EPP can be extended and adapted to work on Finite State Machines and used to find bugs automatically in hardware accelerators generated with High-Level Synthesis. Section 9.4 outlines an additional compression strategy that can be composed with EPP to further reduce the memory footprint of the *Control Flow Checkers* on FPGA up to two orders of magnitude compared to state-of-the-art. Finally, Section 9.5 provides a detailed description of the architecture of the *Control Flow Checkers*.

Part of the material in this chapter was published in an international peer-reviewed journal [24]: P. Fezzardi, M. Lattuada, and F. Ferrandi. Using Efficient Path Profiling to Optimize Memory Consumption of On-Chip Debugging for High-Level Synthesis. *ACM Transactions on Embedded Computing Systems*, 16(5s):149:1–149:19, Sept. 2017.

9.1 Motivation

On-chip debugging is one of the fundamental components of every full-fledged development environment for FPGA applications. Indeed, in hardware debugging, there are some faults that only exhibit on-chip: damaged gates, environmental interference, power supply noises, and in general bugs involved in interfaces with external components. Moreover, High-Level Synthesis is increasingly used for system-level design and to integrate black box Intellectual Property (IP) blocks and hand-written components. IPs provided by vendors may not have been tested for some corner cases of the end users, and hand-written HDL may yield different results in simulation and after synthesis [36] [54]. This scenario fur-

ther complicates debugging when High-Level Synthesis is used for system-level design with integration of third-parties IPs, and calls for a system-level methodology to debug HLS-generated systems directly on-chip.

However, the flow described in Chapter 7 only provides offline simulation-based Discrepancy Analysis. At the same time, the definitions of Chapter 5 are agnostic about how the traces are collected for comparison. Hence, this chapter aims to demonstrate mainly three things:

1. that the Discrepancy Analysis methodology described in the thesis can be actually used for on-chip debugging;
2. that this is possible with optimized dedicated components that enclose all the necessary logic, without imposing restriction to front-end or architectural modification during HLS;
3. that the implementation also has advantages compared to other widespread state-of-the-art techniques.

These goals are very important in validating the proposed Discrepancy Analysis technique because their achievement shows that the methodology is sound and mature enough to be adapted to new and different scenarios.

In particular, in this work the focus is on *Control Flow* Discrepancy Analysis. There are two main reasons for this choice. The first is that even with only *Control Flow* it is already possible to demonstrate the feasibility and to show significant improvements compared to state-of-the-art. The second is that the *Efficient Path Profiling* algorithm, used to compress *Control Flow Traces*, is by its nature only applicable to control flow. *OpTraces* should be handled separately and treated with dedicated algorithms that are not described here.

9.2 Efficient Path Profiling for Software

The main idea behind online Discrepancy Analysis on-chip is to automatically generate and integrate into the designs some dedicated components, called *Control Flow Checkers*. These components will be designed to compare the Software Control Flow Traces representing the golden reference with the Hardware Control Flow Traces generated during the execution of the circuit. In this way, the HCFTs do not even have to be stored on-chip. In order to do this, the *Control Flow Checkers* use an approach based on a software profiling methodology, called *Efficient Path Profiling* [7]. EPP is described here and extended in Section 9.3. The goal here is to understand how EPP describes in compact form the control flow paths executed by software, so that it will be easier to understand how it can be used for hardware debugging.

In general, tracing the control flow of designs generated with High-Level Synthesis corresponds to observing the state signals of the Finite State Machines, as seen in Section 5.1. Finite State Machines are typically built starting from Control Flow Graphs, which in turn represent the structure of the high-level specifications. Every Basic Block in a CFG contains a list of instructions that are executed sequentially in software, while edges are branches and loops. In this way, the CFG statically represents all the possible paths of execution of the software at runtime. The dynamic information about executed paths can be collected by means

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

Figure 9.1: Example of source code to be synthesized, with Basic Blocks ids.

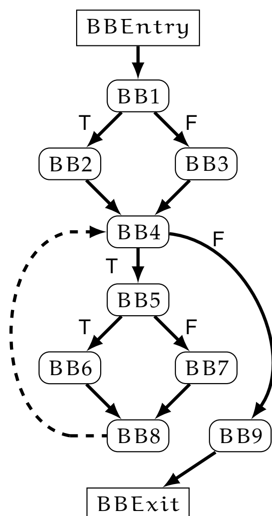


Figure 9.2: Control Flow Graph of the example in Figure 9.1. The dashed arrow is a feedback edge.

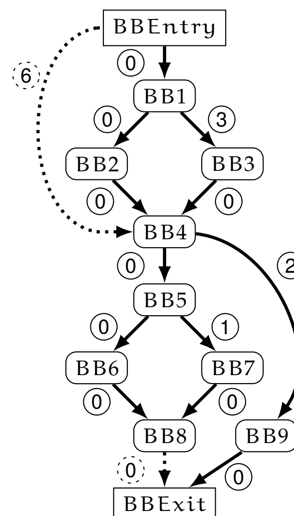


Figure 9.3: Path Graph of the example in Figure 9.1. The dotted arrows are the auxiliary edges.

of Efficient Path Profiling (EPP) [7]. EPP is typically used to collect runtime information about paths in a Control Flow Graph, but in Section 9.3 it will be adapted to hardware generated with High-Level Synthesis.

Intuitively, in software, a path is a sequence of Basic Blocks executed consecutively. Since one specific execution of a function is a sequence of Basic Blocks, it can be efficiently described by means of a path. However, if the Control Flow Graph of a function contains at least one uncountable loop, the number of possible paths which can be executed is potentially infinite, since the loop can be repeated an arbitrary number of times. To overcome this issue, the set of paths which can be extracted from a Control Flow Graph must be restricted, so that the execution trace of a function is described by means of a sequence of paths. Ball and Larus, in their seminal paper on EPP [7], proposed a possible restriction to the paths which can be extracted from a Control Flow Graph and an efficient technique to compute and compress information about them.

Figure 9.1 shows the source of the example used in the rest of this section to describe the Efficient Path Profiling. The corresponding CFG is shown in

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

Figure 9.4: Valid paths of the Control Flow Graph in Figure 9.2.

Figure 9.2. The function contains a loop [78] composed of by the Basic Blocks <BB4,BB5,BB6,BB7,BB8>. The only feedback edge (i.e., the edge which closes a cyclic path with origin in the BBEntry, see [78]) is <BB8,BB4>.

Ball and Larus, in [7], define the valid paths as all the acyclic paths in the Control Flow Graph in the form $<BB_i,\ldots,BB_j>$ such that:

1. BB_i is the BBEntry of the CFG or the target of a feedback edge;
2. BB_j is the BBExit of the CFG or the source of a feedback edge.

This is modeled by building a modified version of the Control Flow Graph: the *Path Graph* (PG). The *Path Graph* can be built starting from the CFG and applying the following steps:

- for every feedback edge $<BB_i,BB_j>$, add an auxiliary edge $<BBEntry,BB_i>$;
- for every feedback edge $<BB_i,BB_j>$, add an auxiliary edge $<BB_j,BBExit>$.

Notice that this procedure is well defined and terminates in a finite number of steps. This is possible because BBEntry and BBExit are not real Basic Blocks, but just placeholders, and, for this reason, they cannot be the origin or the target of feedback edges in the original CFG.

As a result of this construction, the valid paths of the CFG correspond to the paths from BBEntry to BBExit in the Path Graph. The Path Graph derived from the Control Flow Graph in Figure 9.2 is shown in Figure 9.3. Dotted edges are the added auxiliary edges.

Let N be the number of paths in the Path Graph, Efficient Path Profiling uses the number from 0 to $N-1$ to identify them. It also associates a weight $W_{i,j}$ (also called edge increment) to each edge $< BB_i, BB_j >$, so that the identifier of a path is equal to the sum of the weights of the edges which compose it. In Figure 9.3 the edges are labeled with the weights computed with EPP. According to these weights, each path is associated with an identifier from 0 to 8. Figure 9.4 lists the identifiers for the valid paths in Figure 9.3. As an example, the execution trace <BBEntry,BB1,BB2,BB4,BB5,BB6,BB8,BB4,BB9,BBExit> can be compressed in the sequence of paths <0,8> without loss of information. For the details of how path identifiers and edge weights are computed see [7].

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

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

9.3 Efficient Path Profiling for High-Level Synthesis

This section describes how to adapt and extend Efficient Path Profiling to Finite State Machines controlling digital circuits. The aim is to enable online on-chip Discrepancy Analysis for HLS-generated hardware on FPGA, with small memory footprint. In particular, the focus is on control flow, i.e. on the state of the FSMs that control the generated hardware. In this respect, the proposed approach is applicable to all HLS flows that generate functional modules composed of a Finite State Machine and a Datapath. It may not be applicable to other models, like HLS of streaming computations.

Section 9.3.1 describes how the reference traces for the generated design are computed starting from software execution. Section 9.3.2 shows how Efficient Path Profiling is adapted to the debugging of FSMs generated with HLS. Section 9.3.3 illustrates how to guarantee the correct identification of the first bug even when running on concurrent hardware.

9.3.1 Efficient Path Profiling for Hardware Trace Generation

The main idea for the design of the *Control Flow Checkers* is to keep the reference traces small, in order to minimize the memory usage. To this end, Efficient Path Profiling must be adapted to work on FSMs. The advantage of EPP is that a single path represents a list of Basic Blocks. In most cases, storing a path identifier is cheaper than storing the list of identifiers of BBs that compose that same path. To use EPP on Finite State Machines it is necessary to rely on information extracted from the HLS process. During HLS, the Control Flow Graph of the original source code is translated into an FSM. The precise scheduling of the single operations is not really relevant here because the focus is on control flow. What is important is that during the HLS every BB is mapped onto a consecutive list of states in the FSM, like depicted in Figure 9.5. This mapping is called M in the following and it has some useful properties.

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

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

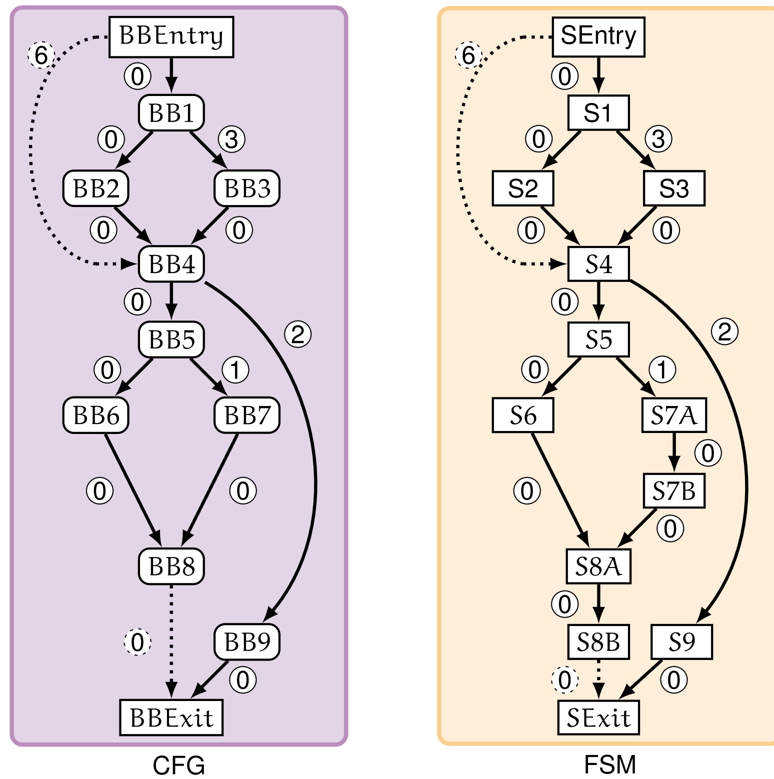


Figure 9.5: Two Path Graphs obtained applying EPP respectively to the Control Flow Graph of Figure 9.2, and to its associated Finite State Machine. Notice the similarities between the graphs, and the equivalence of the edge increments.

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

All these properties make possible to compute the expected list of paths identifiers for the hardware simply starting from software execution. Efficient Path Profiling is used directly on the Control Flow Graph of the software. The code is instrumented to print the list of identifiers of the traversed paths, that can be directly used as a golden reference trace for the generated hardware.

9.3.2 Efficient Path Profiling for Finite State Machines

In Sections 9.2 and 9.3.1 the discussion is focused on Path Graphs, computation of the edge weights for EPP on FSMs, and how to generate the golden reference from software. However, during the actual execution, the hardware follows the Finite State Machine, not the Path Graph. Feedback edges, that were excluded from the computation of the increments, can be taken during execution. In [7], Ball and Larus select a minimal set of edges in the Control Flow Graph and add instrumentations to increment and reset the EPP counter along the edges, including feedback edges. In FSMs, the edges model the state transitions. For effective debugging, the structure of the FSM generated by HLS cannot be altered adding new states or transitions. For this reason, increments, resets, and checks must be scheduled in the existing states. In the following, the discussion is simplified de-

scribing the instrumentations as if they were actually inserted in the FSM. However, bear in mind that they are actually isolated from the FSM and completely encapsulated in the checker component. The FSM only exposes its current and next state to the checker as explained in Section 9.5.

To understand where to insert instrumentations in the FSM, it is useful to make a comparison with software. In software, ensuring that the current execution matches the expected path means ensuring that the path up to that point is correct. To do this, it is enough to check that the counter used to accumulate edge weights matches the identifier of the expected path. Ideally, to have a strong guarantee it would be necessary to check this condition at every cycle, but in practice it is not necessary. Indeed, the only places when the execution path may diverge from the expected are branches and feedback edges. Given that path identifiers are unique, it is enough to check that the current path is correct only on feedback edges and upon the termination of the execution of a function. In hardware, this means that the checks must be performed in states that are destinations of feedback edges, and in terminal states of the FSM.

Final states in the FSM represent the termination of the execution of the associated function. In software, the path counter can be checked after the termination of the function and before returning control to the caller. In hardware, instead, the check is anticipated to the final state itself. This is possible thanks to the fact that a final state of an FSM is a leaf of the graph and has no outgoing edge, which means that the last increment of the EPP counter is computed in the previous cycle and is already available.

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

9.3.3 Detection of the First Mismatch

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

Consider for example the FSM in Figure 9.5, and suppose that a call to another function is scheduled in state S2. The additional idle state is not depicted here

to avoid to overcrowd the picture. The function call is executed conditionally, only if a certain branch is taken. Assume now that the expected path identifier calculated with EPP is <5>, meaning that the expected states executed by the FSM are <S1, S3, S4, S9>. In this situation, according to what explained in Section 9.3.2, the control flow checker would wait to check the execution path until state S9, because no feedback edge is taken. However, if the control flow diverges earlier and it takes the wrong branch from S1, this would result in the execution of S2 instead of S3, triggering the function call scheduled in S2. Given that the called function was not expected to execute, its associated control flow checker would detect a failure, but the root cause is actually a failure in the caller. In this scenario, if the checks are limited to what is described in Section 9.3.2, the detection of the first bug and the automatic identification of the cause would be wrong.

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

9.4 Optimization of Memory Usage

The strategy of adapting EPP for debugging of FSMs explained in Section 9.3 already shows advantages compared to the state-of-the-art (see Chapter 13 for a detailed discussion). This is a consequence of the fact that previous approaches [26] typically encoded control flow traces as a list of states, and then focused on their compression. EPP, instead, represents an entire path with a single identifier. Hence, a single entry in an EPP-encoded trace already represents a compressed list of states without loss of information. This is enough to give better baseline memory usage in most cases (see discussion in Chapter 13), but there are some situations where this is not necessarily true. This happens when the FSM contains a large number of branches and loops, in proportion to the number of states. In such situations two conditions are verified:

1. the total number of valid paths computed by EPP is high, possibly higher than the number of states;
2. the number of times each loop is executed is also high. When both these circumstances are verified, the result is that every single entry in the EPP trace is large (because of the large number of possible paths) and every entry is repeated multiple time in the trace (due to the large number of loop iterations).

In these cases, storing the traces as lists of states does not incur the same penalties as EPP. The reason is that if the number of states is smaller than the number of paths they can be represented with fewer bits. This is especially useful if the bodies of the loops are composed of short lists of states, because a short list of states

may be stored in fewer bits than a single path. This advantage is less significant if the bodies of the loops are long lists of states. However, Goeders et al. [26] show a simple strategy that can be used to compress traces of states specifically in this case. They notice that in the inner loop body the FSM typically traverses a list of states with consecutive identifiers. So they add a fixed number m of metadata bits to every trace entry. These bits are used to store the number of consecutive cycles for which the FSM state simply increases by one. This permits to compress a serial list of states, representing every loop iteration in a single packed entry of the state trace with metadata. The actual size of the metadata is determined with profiling and is fixed for all the FSMs. This approach has a limitation: it directly depends on the encoding of the state signal and the ordering of the states. If the state signal is one-hot encoded, every entry in the state trace is very large and it may need to be re-encoded in a smaller format to reduce the traces. Moreover, to support the compression scheme described in [26] it may be necessary to change the state enumeration. These operations are not necessary with EPP encoding.

EPP does not suffer from these drawbacks because the edge increments are not dependent on the FSM encoding or ordering. However, the problem of compression of multiple iterations of loops with high path weights is still relevant. A loop iteration is represented by a single entry in the EPP trace. Hence, multiple iterations of the same loop are lists of repeated entries, each one representing an iteration. A simple way to compress this kind of traces it to add a fixed number of metadata bits to every trace entry. For every entry in the EPP trace, the metadata represents the number of times the path is repeated after the first time. In this way, every entry in the EPP trace can represent up to 2^k iterations of the same path, where k is the number of bits used for the metadata. Moreover, the determination of the optimal value for k is determined before the generation of the control flow checker. This is a consequence of the fact that the proposed approach aims at finding bugs arising from a predefined input sequence. In this way, once the uncompressed reference traces are generated starting from software, the compression ratio is evaluated as a function of k to generate the control flow checker with the best value for k to minimize memory footprint. This process is performed separately for each checker, so that the optimal value for k is determined for each FSM, depending on its intrinsic characteristics and on the specific trace that must be checked. With this procedure, it is possible to greatly reduce the memory usage on FPGA as discussed in Chapter 13.

9.5 Architecture of the Control Flow Checkers

A dedicated instance of control flow checker is created for the FSM of every function. All the functionalities described in Sections 9.3, and Section 9.4 are implemented in a single component. The checker is separated from the FSM, which is not altered by the instrumentation. The only signals used by the checker are the input and output signal of the state register of the FSM. They are called respectively `next_state` and `present_state` in the following and they drive all the operations of the checker. However, directly using the `next_state` signal may have timing implications, because it is likely to place the checker on a potential critical path. To avoid problems the inputs of the checker are registered so that all the operations of the checker are executed with a delay of one cycle. This does

not prevent the methodology to correctly identify the first fault, because all the checkers are subject to this delay, guaranteeing that the first mismatch notified outside is actually correct.

Every checker contains a read-only memory, called *trace memory*, initialized with the expected EPP trace. It is a single port memory, accessed with a constant fixed alignment and a registered address: `cur_off`. The value in the *trace memory* at `cur_off` is next entry in the EPP trace, containing the identifier of the next expected path. This identifier is also stored in a register called *prev_trace*, used to check feedback edges that are delayed by one cycle as described in Section 9.3.2. A second read-only memory, called *increments memory*, contains the edge increments and is addressed using the union of `present_state` and `next_state`. The value of the edge increment read from this memory is added at every cycle to a register holding the current EPP counter accumulator. This EPP counter is compared directly with the value in the *trace memory* at offset `cur_off`, for states that are checked directly. The value of EPP counter is also reset on feedback edges while registering its value for the delayed check with the mechanism described in Section 9.3.2. The registered value is compared in the next cycle with the value of `prev_trace`. This completes the checking mechanism.

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

What remains is the notification mechanism, by means of which the checker notifies to the outer world when it finds a mismatch. The checker detects if the mismatch is related to the current state or to a delayed check on a feedback edge. If both kinds of mismatches are detected in the same cycle, the one that is notified outside is that related to the delayed check, because it actually happened in the previous cycle. After the selection, the checker writes on output signals the following three data: a bit asserting that a fault was detected, an identifier determined at design time that uniquely identifies the hardware scope where it was detected, and the offset in the EPP trace where that happened. Using this information, the debug environment running on the host machine can unroll the execution traces, map them on the CFG and the FSM, and backtrack the fault to the original source code using HLS information.

It is possible to estimate the area required for a checker in terms of BRAMs and logic. Let $trace_nbits$ be $log_2(PathMax)$, where $PathMax$ is the largest path identifier computed by Efficient Path Profiling for the checked FSM, and *off_nbits* be $log_2(trace_len)$, where $trace_len$ is the length of the golden reference trace for the checker.

The main contribution to the area increase on FPGA is given by the BRAMs used for the traces. The trace memory must requires $checker_mem_bits$ bits,

where $\mathtt{checker_mem_bits} = \mathtt{trace_len} \times \mathtt{trace_nbits}$ bits. This value can be used to compute the total number NRAM of BRAMs that will be used on FPGA, but the result depends on the size of the memories available on the specific target platform, and on the total number of checkers that are added to the design. To give a worst-case estimation it has to be assumed that every trace needs its own BRAM, hence $\mathtt{NRAM} = \sum_{\text{checkers}}(\mathtt{ceil}(\mathtt{checker_n_bits}/\mathtt{BRAM_size}))$. If the traces are small and the available BRAMs are true dual port memories, the actual number could be lower if two traces are stored in a single memory block. This strictly depends on the device and on how the memories are inferred by the synthesis tools.

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

Summary

This chapter described in detail the *On-Chip Online Discrepancy Analysis* flow, based on the adaptation of a software profiling technique called *Efficient Path Profiling* (EPP). Section 9.1 briefly outlined the reasons that motivate the necessity of this extension. Section 9.2 provided an outline of EPP for software, while Section 9.3 explained the subtleties involved in porting it to hardware debugging. Then, Section 9.4 described a strategy to further compress the Hardware Control Traces on top of the compression already granted by EPP. Finally, Section 9.5 gave an overview of the overall architecture of the Control Flow Checkers that are embedded in the HLS-generated designs when using *On-Chip Discrepancy Analysis*.

This concluded Part II of the thesis, which was focused on the definition of the whole proposed methodology. Part III will now focus on the evaluation of *Discrepancy Analysis*, providing results obtained on different benchmarks to measure different features of the various flavors of *Discrepancy Analysis* described in Part II.

PART III

EXPERIMENTAL RESULTS

This part describes the results obtained during the experimental evaluation of the proposed methodology.

Chapter 10 introduces the experimental setup, explaining how Discrepancy Analysis were integrated into the High-Level Synthesis flow, as well as the other tools used for the evaluation and the benchmark.

Chapter 11 provides a wide and detailed discussion of the detected classes of bugs, involving different steps of High-Level Synthesis process and different flavors of Discrepancy Analysis in different scenarios.

Chapter 12, and Chapter 13 then focus on the performance, the coverage and the other advantages provided by Discrepancy Analysis. Chapter 12 concentrates on Simulation-Based Offline Discrepancy Analysis, whereas Chapter 13 is focused on On-Chip Online Discrepancy analysis.

Finally, Chapter 14 summarizes the results, pointing out the achievements of this work, its limitations, and discussing some of the possible future direction of research.

10

Experimental Setup

This chapter describes the general setup used to evaluate the methodology proposed in this thesis. Section 10.1 discusses the HLS framework used for the reference implementation of the two workflows described in Chapter 6. Section 10.2 lists the different sets of benchmarks and tools used to stress-test the implementations of the different flavors of the approach: simulation-based online Discrepancy Analysis, bug detection for multi-threaded code, Address Discrepancy Analysis, and on-chip online Discrepancy Analysis for Control Flow.

10.1 Integration with High-Level Synthesis

The reference implementation for the automated bug detection based on Discrepancy Analysis was built upon the BAMBU High-Level Synthesis compiler. BAMBU is developed at Politecnico di Milano as part of the *PandA Framework for Hardware/Software Codesign* (`https://panda.dei.polimi.it`). It is Free and Open Source Software, released under the GPLv3 License. The BAMBU compiler is modular so that it can be easily extended with custom passes for specific optimizations. It features a novel memory architecture that supports a wide range of `C` constructs, limiting as much as possible code rewriting and simplifying system-level integration. It accepts inputs in `C` and `C++`, and it can generate Verilog and VHDL designs for a range of target FPGA devices from different vendors: Xilinx, IntelFPGA (former Altera), and Lattice. It also allows to seamlessly add support for new boards if necessary.

The choice of BAMBU, among other available alternatives, as a starting point for the research effort described here, has various motivations.

First of all, its modular design and its open source license were the most suitable to understand its inner operations, and to extend the compiler passes to extract the information necessary to enable Discrepancy Analysis. At the same time, BAMBU can generate accelerators with an advanced memory architecture and it performs several state-of-the-art optimizations, resulting in a quality of the obtained results (in terms of frequency and area) that is comparable to production ready and widely used industrial HLS tools (see `https://panda.dei.polimi.it/?page_id=317`). In addition, BAMBU provides support for a subset of the OpenMP pragmas, which allowed testing the reference implementation for bug detection on multi-threaded code, like explained in Section 7.3. The advanced memory model, instead, was one of the key features necessary to test the *Address Discrepancy Analysis* described in Chapter 8. The modular design of BAMBU's HLS flow also played a key role in the extension for On-Chip Discrepancy Analysis described in Chapter 9. Indeed, to generate the fine-tuned hard-

ware checkers described, it is necessary to exploit information gathered with the execution of the instrumented software. The modular design allowed to integrate all the different components and flavors of the Discrepancy Analysis directly into the HLS flow, providing a consistent user experience. It also allowed making all the HLS information accessible to the bug detection algorithm, to guarantee visibility on compiler temporary variables, to hide the internals from the users, and to manage complexity for them.

Another important feature in the choice of an HLS tool for the reference implementation was the integrated environment for testing and co-simulation available in BAMBU. Like many commercial solutions, BAMBU allows to run the original high-level source code and to simulate the generated HDL to check that the results are the same. This workflow is seamlessly integrated with HLS and natively supports many different commercial simulators, as well as open source alternatives. The availability of this workflow was a nice-to-have because Discrepancy Analysis can be naturally implemented as an extension to it. The co-simulation workflow checks only the return values, while the Discrepancy Analysis flow for automated bug detection compares the complete traces generated by the software with the hardware traces obtained with simulation.

Clearly, the reference implementation for the BAMBU compiler described in this work contains some details that are specific to that tool. However, a considerable effort has been made in this thesis to avoid considerations and implementation details that are strictly related to BAMBU, and to make the approach suitable to be used with other tools. Sometimes, to test some scenarios supported by a commercial tool and not by BAMBU, the necessary HLS information have been gathered manually from simple designs generated with the commercial tool itself. Then the bug was manually inserted in the design and the bug detection algorithm run step-by-step, to demonstrate that the bug would have been spotted with Discrepancy Analysis also in that scenario. When this happened, the details of the benchmarks, the injected bugs, the specific optimization not supported by BAMBU, and the precise steps used for the evaluation of the results are explained in detail in the discussion.

The complete methodology described in this thesis, composed of Discrepancy Analysis, Address Discrepancy Analysis, and On-Chip Discrepancy Analysis, has been fully integrated into BAMBU. The complete flow based on simulation for offline Discrepancy Analysis has already been released with BAMBU and is available for download with the version 0.9.5 of the framework at `https://panda.dei.polimi.it`. On-Chip Online Discrepancy Analysis will be released with the next version of the framework.

10.2 Experiments and Benchmarks

Several experiments have been carried out to evaluate how the proposed methodology performed in different scenarios. In particular, the investigation was aimed at the following main goals:

- identify at a high-level what classes of bugs the Discrepancy Analysis is able to detect, both in the HLS process, in external libraries of third-parties components, and in hand-written code provided by users;

- provide a coverage metric for the checks performed by Discrepancy Analysis at operation level;
- measure the performance and the overhead of the Discrepancy Analysis, as well as other positive consequences on the whole debugging experience;
- detect potential false positives or false negatives and measure their impact and frequency in realistic scenarios.

To examine these points, different sets of benchmarks have been used to measure different properties in the different scenarios described in Part II. Four main families of experiments can be identified, each of them focused on analyzing a different aspect of Discrepancy Analysis.

1. Offline bug detection with Discrepancy Analysis based on simulation:
   (a) on serial code;
   (b) on multi-threaded concurrent code;
   (c) on pointers and memory accesses with Address Discrepancy Analysis.
2. Online bug detection with On-Chip Discrepancy Analysis on control flow.

The remainder of this section describes the experiments, the testing procedures and the tools used to evaluate these four scenarios.

10.2.1 Experimental Setup – Discrepancy Analysis for Serial Code

The principal set of benchmarks used to evaluate Discrepancy Analysis is the *CHStone* benchmark suite [33]. The *CHStone* benchmarks are 12 self-contained C programs, selected as representative examples of idiomatic code used in High-Level Synthesis. They are well-known and a de-facto standard for benchmarking in the High-Level Synthesis community. These benchmarks have been initially used to evaluate the baseline version of Discrepancy Analysis, without the support for multi-threaded code, and for pointer operations. They were also used in the evaluation of the extended versions of Discrepancy Analysis, but often together with other test cases that were more meaningful for the specific needs of the investigation.

All the *CHStone* benchmarks have been translated in Verilog with the default configuration of BAMBU, targeting a Xilinx Zynq-7000 xc7z020-1clg484, with a target frequency of 100MHz. Other configurations and target devices were also tested, to verify the robustness of the approach, but no noticeable differences have been observed on the effectiveness of the methodology, nor on the quality of the results, nor on the performance. The results reported in Chapter 12.1 refer to this baseline configuration.

To evaluate the simulation-based Discrepancy Analysis, the designs were not synthesized and executed to FPGA. The hardware traces results have been generated with simulation, using ModelSim SE-64 10.5 from Mentor Graphics, but there is nothing specific to this simulator in the process. During some preliminary tests, other commercial and open source simulators supported by BAMBU's co-simulation flow have been successfully employed, without affecting the bug detection capabilities of the Discrepancy Analysis.

For the generation of the software traces, the instrumented source code has been compiled and executed with GCC, version 4.9.

10.2.2 Experimental Setup – Discrepancy Analysis for Multi-Threaded Code

The *CHStone* benchmarks are constituted by plain serial code, so they were not suitable to evaluate the methodology with multi-threaded code. This scenario was evaluated using 7 benchmarks that have already been used to evaluate HLS of OpenMP programs by Choi et al. [17] with *LegUp*.

- *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 [33], 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 2-dimensional grid is visible from the source.
- *Mandelbrot* (mb): an iterative mathematical benchmark which generates a fractal image.
- *MCML* (mcml): light propagation from a point source in an infinite isotropic medium.

Some of the benchmarks as they were used in [17] contained a mix of OpenMP and `pthreads`, but they have been adapted to only use OpenMP for the purposes of this work. Again, BAMBU was used with its default parameters to generate Verilog designs for all the benchmarks, simply adding the `-fopenmp` flag to enable OpenMP. The default target device is a Xilinx Zynq-7000 xc7z020-1clg484, with a frequency of 100MHz. The integrated co-simulation flow was executed before using Discrepancy Analysis, to ensure that the generated hardware was working properly. Then, different kinds of bugs were manually injected to see if they could be detected (see Chapter 11). The automated bug detection was also executed on the unmodified multi-threaded designs, to check for false positives and to measure its overhead.

Also in this case, the simulation used to generate HW traces is cycle-accurate and it has been performed with ModelSim SE-64 10.5 from Mentor Graphics, and the compiler used to compiler the instrumented code for software trace generation was GCC-4.9.

10.2.3 Experimental Setup – Address Discrepancy Analysis

The analysis of Address Discrepancy Analysis required a broader perspective. Given that different HLS compilers provide different schemes for memory partitioning, the approach was evaluated on three different tools, to ensure the neces-

```
int w(struct sockq *q, void *src, int len) {
    char *sptr = src;
    while (len--) {
        q->buf[q->head++] = *src++;
        if (q->head == NET_SKBUF_SIZE)
            q->head = 0;
    }
    return len;
}
```

Figure 10.1: A C function with pointer operations not supported by *CTool*.

sary level of generality. The experimental results have been evaluated with the following compilers, with support for C pointers:

- BAMBU;
- *LegUp* [15], based on LLVM [48] and developed at the University of Toronto;
- a recent version of a production-ready commercial HLS tool targeting Xilinx FPGAs, referred in the following as *CTool* (the license does not allow to disclose the name).

All of them are able to synthesize pointers, with different memory partitioning and allocation schemes. Most of the tests were carried out with BAMBU, but some unsupported partitioning schemes were tested with *LegUp* and *CTool*.

For Address Discrepancy Analysis, the evaluation involved two groups of benchmarks: the *CHStone* HLS benchmark suite [33] and the GCC *C-torture* test suite [25]. The first establishes a common baseline for all the tools, but it has a big drawback for the scope of the research: it has no complex pointer operations. They are complicated enough to make alias analysis not trivial, but they really do not try to push the limits of what can be done with pointers in valid C code. This is not useful when testing the Address Discrepancy Analysis because the focus is on seeing how it behaves with exotic pointers manipulation in C. This is the main reason behind the decision to use also the GCC *C-torture* tests. These are a large set of self-contained C programs, specifically designed to exercise corner cases of a standard-compliant C compiler, including a number of uncommon things with pointers. From preliminary trials, it turned out that only 216 of such tests were involving pointer operations. The analysis is restricted to this subset. On 56 of them, *CTool* failed to complete the HLS process. An example of a C code snippet that could not be handled is shown in Figure 10.1. Notice that from the programmer standpoint the use of pointers in this function it is not particularly strange. This kind of syntax is common practice in embedded C code, but still many commercial tools have problems in handling pointer casts and other similar operations. For this reason, for the extensive tests with the *C-torture* tests, the choice of HLS tool still fell on BAMBU, which also has an advanced approach to memory allocation [67] and it implements fairly complex frontend optimizations [49].

The main obstacle to the evaluation of *Address Discrepancy Analysis* is that some of the information for the construction of the *Address Space Translation Space* (ASTS) must be extracted from the HLS compiler. This is also necessary to understand how to design the implementation dependent `decodeHW` function de-

scribed in Section 8.2. For BAMBU and *LegUp* this is not a real problem because of their open source licenses. This allows modifying the memory allocation pass of these compilers to obtain the data. A preliminary feasibility study on both these open source tools showed that this was possible for both, with the same fundamental approach. The full-fledged implementation of Discrepancy Analysis was created only for BAMBU, because it has a very complex memory model, allowing tests in more challenging cases. For this reason, the results on bug detection and on coverage metrics reported in Sections 11.2 and 12.3 are strictly related to BAMBU. For *LegUp*, the algorithm was applied manually on the *CHStone* benchmarks, to check that the methodology was actually portable even without deploying the full-fledged automated flow.

For *CTool* some additional work was necessary to build the ASTS and the `decodeHW` function because the HLS flow could not be altered. Some of the CHStone benchmarks that the tool was able to synthesize were compiled using memory partitioning directives. The chosen designs were test cases for which *CTool* generates correct address decoding because, to infer the ASTS and the `decodeHW` function from the HDL, one has to be sure that they were correct before injecting the bug. For this reason, the analysis of *CTool* was restricted to examples where the generated designs were correct and passing all the functional tests. This allowed manual analysis of the generated HDL to build the correct ASTS and the `decodeHW` function. After building them, the HDL of the address decoding logic generated by *CTool* was altered manually to introduce bugs. This operation showed that it is possible to use the correct ASTS and `decodeHW`, built in advance, to spot a hypothetical bug manually applying the bug detection algorithm to the fault-injected design. Obviously, the problems analyzed in this way are a subset of all the possible cases. Nevertheless, the results are encouraging because they show that the method is successfully applicable even to commercial HLS flows. In the experiments with *CTool* the investigation was done manually, but with the access to the source code, it should be easy to adapt the tool to implement an automated debug flow as described for BAMBU. This has been valuable to show that the methodology can target commercial HLS tools and that it can handle memory options not available in BAMBU, like array partitioning.

In all the cases, with BAMBU, *CTool* and *LegUp*, the simulation used to generate HW traces is cycle-accurate and it has been performed with ModelSim SE-64 10.5 from Mentor Graphics, and the compiler used to compile the instrumented code for software trace generation was GCC-4.9.

10.2.4 Experimental Setup – On-Chip Online Discrepancy Analysis

To evaluate Online Discrepancy Analysis On-Chip, an implementation of the workflow described in Chapter 6.3 has been integrated into BAMBU. Two main changes have been applied to the framework:

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

The methodology flow has been tested on the *CHStone* benchmarks [33]. The

benchmarks have been compiled to Verilog using BAMBU with the default configuration of the *PandA* framework. The only modification was to target a Stratix V device (5SGXEA7N2F45C1) from Intel (former Altera) with a target frequency of 200MHz, to provide a better comparison with state-of-the-art [26]. Quartus Prime Standard Edition 17.0 was used for synthesis, to generate the bitstreams for the designs with and without the control flow checkers. All the results concerning area, frequency and power consumption of the checkers have been computed by the synthesis tool, after place-and-route to increase the accuracy of the results as much as possible. They are reported in detail in Chapter 13.

Summary

This chapter opened the discussion of the results, starting with the experimental setup. Section 10.1 described BAMBU, the HLS framework used for the reference implementation of *Discrepancy Analysis*, along with the features that have led to its choice among other available alternatives. Then, Section 10.2 provided an overview of the different benchmark suites used later to evaluate the methodology, as well as the tools used in different stages of the evaluation. This should set the frame to help the reader in the understanding of the reported results. Some of this information will be repeated when necessary to avoid confusion.

Chapter 11 will now provide a complete overview of all the classes of bugs that can be found with *Discrepancy Analysis* in different scenarios: bugs in standard designs generated with HLS from serial C programs, bugs involving addresses and memory accesses, designs generated from multi-threaded specifications, and bugs detected online on-chip.

11

Detected Bugs

This chapter describes the bugs that the proposed methodology was able to find during extensive experiments. Section 11.1 describes bugs detected with the offline simulation-based Discrepancy Analysis on hardware generated from single-threaded programs, while Section 11.2 discusses the additional faults that were detected with the extended Address Discrepancy Analysis. Section 11.3 focuses on bugs coming from High-Level Synthesis of multi-threaded code. Finally, Section 11.4 analyzes the misbehaviors that can be detected with online on-chip Discrepancy Analysis.

Each of these categories is described in detail, providing a high-level discussion of the different classes of bugs detectable with the approach, as well as practical examples found during the experiments. Some of the discussed faults were actually found in the RTL generated from BAMBU, and are now fixed in the distributed version. Others have been manually inserted to test the capabilities of the methodology.

11.1 Bugs Detected with Simulation

This section analyzes the kind of bugs that can be detected The bugs detected with the offline simulation-based Discrepancy Analysis described in Chapter 7. Two classifications of such bugs are provided: the first is focused on the root cause that originates the bugs, the other is concerned about how these bugs affect the generated hardware design.

Classification According to the Root Cause

Bugs detected with the offline simulation-based Discrepancy Analysis can be roughly divided into three classes, according to the their root cause:

(1) bugs already present in the original C specification;

(2) bugs introduced by the HLS tool;

(3) bugs introduced using a library with flawed hardware components for HLS.

In particular, among the bugs in group (1) already present in the original C code (1), those that can be detected are due to *unspecified behavior* or *undefined behavior* in the C standard. In several places of the C standard, the definition of the semantics of certain operations is left *unspecified*, like for example the order of evaluation of the arguments of a function, but also others. The goal is to leave the freedom to implementors to choose the most efficient behavior on different platforms. Where possible, the standard defines a set of allowable behaviors for

a given instance of *unspecified behavior*. These define the non-deterministic aspects of the abstract machine that defines the C execution model. They may also depend on compiler options. On the other hand, *undefined behavior* regards operations for which the C standard does not define any precise semantics. As a consequence, an implementation is considered standard-conforming whatever it does in such cases. This allows compilers to treat *undefined behavior* with special measures, possibly enabling very aggressive optimizations.

At the same time, there are cases of *unspecified behavior* where C most compilers do the same '*expected*' thing, so that programmers have started to silently rely on that behavior without even knowing it. This issue is analyzed in a work by Memarian and Sewell [52], and it represents an issue in HLS, given that the '*expected*' thing to do on FPGAs is not necessarily the same as on CPUs or what programmers expect. An example is represented by non-initialized variables or integer functions with `return` statements without value. These are allowed by the C standard, but they cause non-initialized signals to be set to "Z" in Verilog. Others are just real bugs caused by a wrong implementation of the specifications in the original C source. In this case, the Discrepancy Analysis does not discover any mismatch between the high-level source code and the resulting HDL. However, given that this kind of bugs are already present in the original code, all the well-known software debugging techniques can be used to find them, without involving the HW.

Bugs in categories (2) and (3) are becoming increasingly important in the HLS field, for two different reasons. On the one hand, bugs introduced by HLS tools (2) are very subtle to identify and analyze for hardware designers that use HLS without knowing the internals. For this reason, it is critical for the success of HLS to ensure that they occurrences are almost negligible, possibly non-existent. Due to recent advances in HLS and industry investments, HLS is thriving and tools are becoming increasingly complex and using smart optimizations. In this scenario, *Discrepancy Analysis* can be a valuable tool to help developers of HLS frameworks to automatically identify bugs in their tools at early stages, before shipping flaky implementations to customers. On the other hand, HLS is increasingly being used for system integration, gluing together different components, some of which might not be designed with HLS. For this reason, bugs in category (3) are becoming frequent, and the ability to identify them automatically using *Discrepancy Analysis* can boost productivity for hardware designers.

Classification According to the Effects on the Hardware

Bugs in categories (2) and (3) tend to affect the final designs in various different ways. However, they can be classified according to their impact on the design, which is typically one of the following:

(A) bugs in HW components used to implement operations;

(B) bugs in the FSM controller logic;

(C) bugs causing the design to loop or hang;

(D) errors in the interconnection between components;

(E) bit flips due to aggressive optimizations.

Bugs in all these categories have been successfully detected by Discrepancy Analysis. Some of them were manually inserted for testing purposes, while others were actually found in the BAMBU HLS compiler. An out-of-bound bug was found in the `mips` benchmark in the *CHStone* HLS benchmark suite [33] (version 1.11_150204 and previous). Bugs of group (A) and (E) are the more frequent and they are detected with per-operation accuracy. When a mismatch occurs, the tool shows the position of the failing operation in the original C along with the mismatching signal and the timestamp. The failing operation may not be present in the original code, for instance if it was inserted by compiler optimizations. In this case, the automated bug detection shows the information on the instrumented code after the optimizations. Bugs in the logic of the generated controller (B) can be due to wrong state optimizations of the FSM. They can cause a mismatch in the CFTs or bugs of type (C). In the latter case, the simulator can be set with a maximum number of cycles to simulate. Then *Discrepancy Analysis* is performed on the partial traces. In this way, the same method can be used to find bugs which normally would cause the design to hang or loop. Finally, bugs of categories (D) and (E) are actual compiler bugs. It is worth to remark that these experiments proved that with this approach it is possible to spot bugs which are not visible outside the design. Moreover, it shows that Discrepancy Analysis can be also very useful to test HLS compiler implementations.

11.2 Bugs Involving Addresses

Applying extensively the described approach to BAMBU it was possible to find several bugs involving pointer operations. Interestingly, the bugs detected with this method are not always strictly caused by errors in memory allocation. Instead, they can be generated by problems in other steps of HLS. The one thing they have in common is that they affect in some way the address decoding logic of the generated circuit, causing a wrong address to be computed at some point. Here is an exemplifying but not exhaustive list of the affected steps in HLS, with some of the found bugs.

Compiler Frontend

Bugs due to wrong static analysis or manipulations of the IR, before the actual HLS takes place. Among them, a compiler pass in BAMBU performs bit-width static analysis to reduce the bits necessary for addressing memories. This step was buggy and sometimes the number of bits that the tool required to be necessary to represent addresses was too high or too low. In both cases, the effect was that wrong values were used to address the memory, causing bugs that propagated to the rest of the design.

Scheduling

Problems due to wrong scheduling of operations in the FSM. They include wrong reordering of operations due to missing dependencies, bad scheduling due to wrong computations of operations' execution times and others. In some cases, BAMBU's frontend lost information about data dependencies among operations. As a consequence, the scheduling step in HLS decided that an address could be

computed in advance, but the data used for the computation was actually not ready to use, again generating wrong addresses.

Memory allocation

Instantiation of memory modules with wrong characteristics, size or latency. This happened with BAMBU and it caused different kinds of problems. When the memory was too small, some data could be lost writing it to an out-of-bound address. It also happened that the HW tried to read data from an out-of-bound address, causing the design to hang, waiting for a reply from memory that never happened. When the memory was too large, the offset calculation in address decoding was wrong. This caused memory accesses at wrong locations, reading wrong data or writing them in the wrong place. Finally, when the expected latency was wrong, the HW was using data before the memory replied.

Interconnection

Wrong connection of wirings, causing malfunctions. For instance, the same bug described at point (1) affected the generation of the interconnection. Thus, the address bus had the wrong width, causing wrong addressing.

All these bugs were properly detected and isolated in BAMBU by the approach, without human intervention. The data provided by the *Discrepancy Analysis* engine allowed to identify the cause and to fix the HLS compiler. Positively, the approach was able to treat bugs causing the designs to hang or loop forever because the simulation could be interrupted to perform *Discrepancy Analysis* up to a certain point in the execution. Another important thing to stress is that most of the memory bugs detected with *Discrepancy Analysis* were also causing errors in variables that did not represent addresses. Clearly, if a READ loads data from the wrong location, it is likely to get them wrong. Without the Address Discrepancy Analysis, it would not be possible to know if the problem was the address or the data in the memory itself. The GCC *C-torture* tests were very valuable in this phase, because the CHStone benchmarks did not trigger any of these bugs.

11.3 Bugs in Multi-Threaded Programs

The bug detection algorithm was tested manually inserting three different kinds of bugs. The *first* class is composed of bugs located in a *single hardware thread*. The *second* class of bugs involves *communication between threads* via shared memory. Bugs in the *third* class were caused by *missed or multiple executions of tasks*.

Bugs located in single hardware threads

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 and every single thread with the same accuracy of the single-threaded version, even if the 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 Section 8).

The approach can also isolate bugs in libraries of external components used as elementary operators in HLS.

Bugs involving communication between threads

The extended Discrepancy Analysis detects situations where thread accelerators read or write 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. The Discrepancy Analysis 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 12.2.3.

Bugs 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.

Remarks

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 7.3 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.

11.4 Bugs Detected On-Chip

On-chip Discrepancy Analysis has shown to be able to detect different kinds of bugs compared to the simulation-based approach. As shown in Figure 11.1, the set of bugs detected on chip partially overlaps with the set of those found with simulation. The set on the left represents the bugs detected with offline simulation-based Discrepancy Analysis, while the set on the right represents the bugs detected with online on-chip Discrepancy Analysis. It is interesting to analyze the overlapping and disjoint portions of sets more closely. During the experimental evaluation, bugs belonging to these three subsets have been manually crafted and injected in the designs, to assess the real capabilities of Discrepancy Analysis.

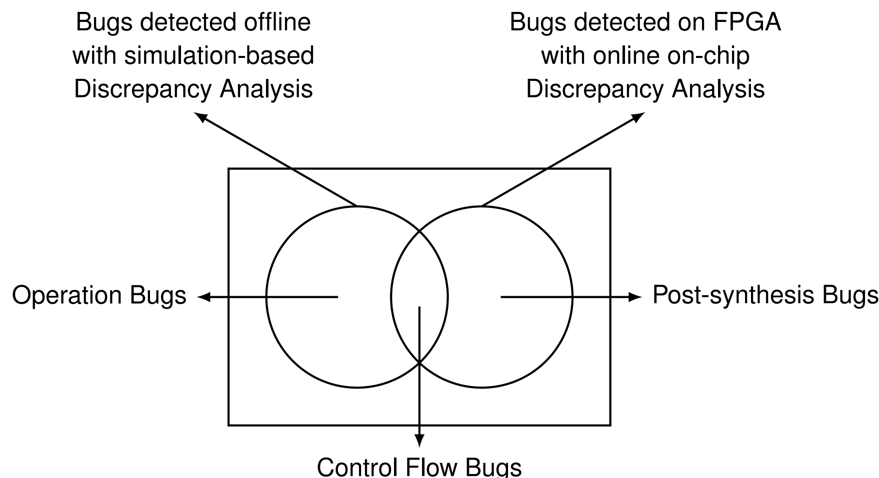


Figure 11.1: Venn diagram of the classes of bugs detected with Discrepancy Analysis. The set on the left represents the bugs detected with offline simulation-based Discrepancy Analysis, while the set on the right represents the bugs detected with online on-chip Discrepancy Analysis.

First, not all the bugs that can be detected with simulation are also visible on chip. This is not surprising, and it is a direct consequence of the fact that the hardware checkers proposed in Chapter 9 only perform online Discrepancy Analysis on control flow. For this reason, all the bugs directly involving only results of operations are not detectable. However, this restriction is not a theoretical limit of the On-Chip Discrepancy Analysis per se. The only reason of this limitation is that, in the version showcased here, the checkers only implement Discrepancy Analysis at *control flow level* by design. If the checkers are extended to check single operations this limit could be overcome.

The second thing to notice is that the overlapping portions of the two sets is constituted by control flow bugs. This means that, for what concerns control flow bugs, all the bugs detected with simulation can also be detected on-chip with the same accuracy. This is a direct consequence of the fact that the design of the control flow checkers is based on the same notion of equivalence used for checking Control Flow Traces coming from simulation.

Finally, there are also bugs that can only be detected on chip. These bugs are typically caused either by faulty third-parties IP blocks that may not have been tested for some corner cases of the end users, or by hand-written HDL coding styles that may yield different results in simulation and after synthesis [36] [54]. Another common case when they arise is when designers test in integration on chip some components that have only been tested in isolation, surfacing problems at the interfaces. Both these two scenarios are becoming increasingly common due to component reuse and use of HLS for system integration. Being able to detect such cases is important, and improves the final debugging experience.

Summary

This chapter summarized all the different classes of bugs that can be detected with *Discrepancy Analysis*, involving different steps of the HLS process. First, Section 11.1 focused on the bugs detected with simulation on designs generated from standard serial `C` programs. Then, Section 11.2 provided an overview of bugs in-

volving pointers and memory accesses. Section 11.3 analyzed the bugs detected in designs generated from multithreaded specifications. Finally, Section 11.4 discussed the bugs that can be detected on-chip.

This discussion on the classes of bugs is useful to understand the capabilities of *Discrepancy Analysis*, but it is not enough to give a complete overview of how this methodology can positively affect the debugging experience for circuits generated with High-Level Synthesis. For this reason, Chapters 12 and 13 provide extensive measurements gathered on different benchmark sets with *Simulation-Based Offline Discrepancy Analysis* and *On-Chip Online Discrepancy Analysis*.

12

Simulation-Based Discrepancy Analysis

This chapter discusses a number of experiments aimed at the evaluation of the *offline simulation-based Discrepancy Analysis* described in Chapter 7 and the *Address Discrepancy Analysis* introduced in Chapter 8. In particular, different classes of experiments have been carried on to evaluate different aspects of the methodology.

Section 12.1 focuses on the results obtained with the baseline implementation introduced in Sections 6.2, 7.1, and 7.2. Section 12.2 reports the results obtained on multi-threaded benchmarks, hence focusing on the extension to multi-threaded programs described in Sections 7.3. Finally, Section 12.3 reports data on the extension to pointers and memory accesses explained in Chapter 8.

12.1 Baseline

This section describes the results obtained with the baseline implementation of *Offline Simulation-Based Discrepancy Analysis*, without support for multi-threading nor for pointers and memory accesses. This is useful to realize the general behavior of the approach, in terms of performance, granularity of the checks, and other advantages that it brings to the overall debugging experience.

Section 12.1.1 reports results on the performance of the bug detection algorithm; Section 12.1.2 discusses the granularity of the checks performed by *Discrepancy Analysis*, including temporary variables introduced by the compiler for optimizations; Section 12.1.3 describes other advantages descending from the automated signal selection performed, that contribute to improving the overall user experience.

12.1.1 Performance

Figure 12.1 shows the time overhead of our the *Discrepancy Analysis* bug detection algorithm, compared to the execution time of the simulation of the design under

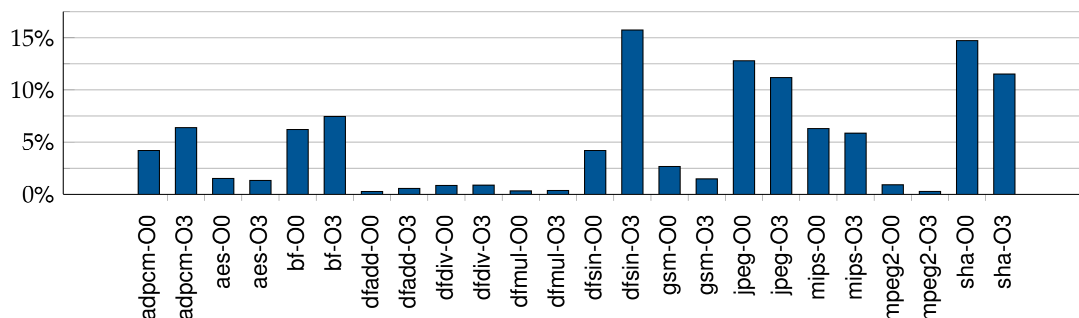


Figure 12.1: Time overhead of *Discrepancy Analysis*, compared to simulation.

test. The simulation is cycle-accurate and it was performed with the ModelSim SE-64 10.5 simulator. The names of each benchmark are reported below the related bar, along with the optimization levels. The two levels evaluated here are -O0 and -O3, depending on the flags passed to the compiler.

From the figure, it is clear that the execution time overhead is negligible: just above 15% in the worst cases, but much less in most of the others. This is even more impressive considering that normally the user would have to explore the designs manually, reconstruct the optimizations performed by the HLS process, backtrack the HDL description to the original high-level source code, and finally analyze the signals manually, without good knowledge of the circuit, to identify and locate the first bug occurred during the execution. In this sense, if the overhead of the bug detection is compared to the time it would take to a human designer to perform the same tasks, the little overhead to pay for *Discrepancy Analysis* is a great improvement.

The data reported in the figure were gathered executing bug detection on designs that were free of bugs, to measure the full worst-case runtime of the algorithm. In case a bug is present the overhead is much less, about 10 to 20 times less depending on the location and the timing of the first mismatch.

The data reported in Figure 12.1 do not include support for multi-threading, nor for automated bug detection of pointers. Handling these special cases requires more work and computation, hence the overhead is higher. Detailed data are reported in Sections 12.2 and 12.3.

12.1.2 Coverage

Another interesting result to measure is the granularity of the checks performed by *Discrepancy Analysis* during the automated bug detection. In order to measure it, it is necessary to define a good coverage metric.

In this respect, it is important to make a few considerations about what kind of coverage it is meaningful to measure. There are two main tricky points here:

- the language for which the coverage metric is defined;
- whether the coverage metric has to be static or dynamic.

The first point is specific to the fact that *Discrepancy Analysis* is a methodology for automated bug detection *in circuits generated with HLS*. Dealing with HLS means that there are two languages involved in the design process: a low-level language, i.e. the HDL description of the circuit; a high-level language, i.e. the input language of the HLS tool. Given that *Discrepancy Analysis* aims at providing useful information to HLS users without requiring deep knowledge of the HLS process itself, the most natural choice here is to define a coverage metric on the high-level source code.

The second point, whether the coverage metric has to be static or dynamic, depends on the nature of the bug detection operation. Generally speaking, a static coverage metric measures how many of the overall static instructions in a program can be checked with a given methodology. On the other hand, a dynamic coverage metric measures how many of the overall static instructions in the same program are actually checked during a given execution of a program (or a set of executions with different inputs). In general, dynamic coverage metrics are

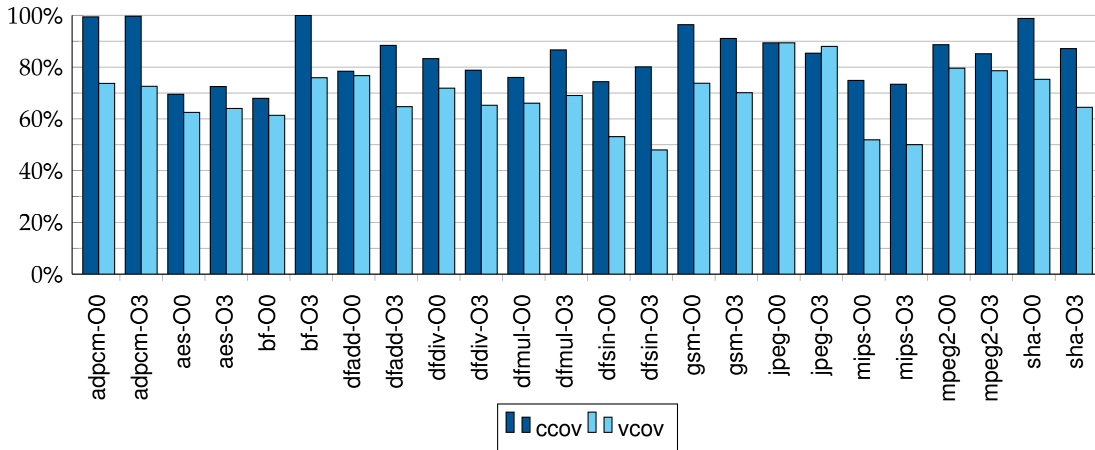


Figure 12.2: *Statement coverage* for the *CHStone* benchmarks. The coverage was computed for C (ccov) and the Verilog designs generated with BAMBU with different optimization levels (-O0, and -O3).

useful to understand how effective is a technique to test exhaustively a design. However, testing exhaustively a design is not the goal of *Discrepancy Analysis*. *Discrepancy Analysis* aims at providing a fast and effective way to automatically detect bugs when they arise. For this reason, it is orthogonal to other techniques that strive to provide exhaustive testing for hardware designs. But exactly for this reason, it makes little sense to use a dynamic coverage metric of this kind to measure the effectiveness of *Discrepancy Analysis*- A static coverage metric is more suitable for this goal.

The static coverage metric used in the rest of this chapter is called *instruction coverage* (icov) and it can be defined as follows:

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

It measures the percentage of static operations in the elaborated C program that can be checked with the *Discrepancy Analysis*. At first sight, it may resemble *statement coverage*, which in general is defined as follows:

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

In the remainder of the thesis, the notation ccov is used for C statement coverage and vcov for Verilog statement coverage. For the sake of completeness, Figure 12.2 reports the dynamic *statement coverage* for C ('ccov') and Verilog ('vcov'). For C, ccov was computed with gcov, the coverage tool included in GCC-4.9. For Verilog, vcov was evaluated with simulation, using ModelSim. The data show that the tests are not covering the whole design, nor the original program. This is expected, since the data set on which the *CHStone* operate is fixed and it was not designed to provide full coverage. Currently, *Discrepancy Analysis* does not consider how the inputs for the design under test are generated, and the default dataset was used for all the experiments with the *CHStone*. However, the problem of how to tune the input test is interesting, especially because there is no straightforward relationship between coverage in C and in Verilog. The topic is orthogonal to what is described here. It is very vast and it deserves a separate analysis.

Despite the similarities, *instruction coverage* is not equivalent to the *statement coverage* in the original C code nor in the generated HDL. First of all, icov is a static metric, while *statement coverage* is dynamic because it measures the number of statements that are actually executed at runtime. Instead, icov measures at compile time the number of high-level statements whose results can be checked by *Discrepancy Analysis*, even if at runtime such instructions are not executed, depending on the test input. Secondly, icov has a finer granularity, because it considers the values of the intermediate sub-expressions in statements separately. In this way, it is possible to check variable assignments, but also intermediate values assigned in composite statements. This metric measures the granularity of the checks. The goal is to show that *Discrepancy Analysis* is able to instrument and check a large majority of the operations, independently of the fact that they are actually executed at runtime.

Notice that the *instruction coverage* does not only includes assignment statements, but also control flow instructions. Branch statements, function calls and return statements cannot be checked directly by *Discrepancy Analysis* because they do not assign variables. However, they can be checked indirectly. In fact, function calls are not directly checked, but the operations in the body of the called functions are. The same holds for the arguments passed by the caller to function calls. Return statements are not directly checked inside the body of the returning function, but the returned values are checked right after their evaluation, before the return. Finally, branch instructions are not directly checked, but the branch condition is checked at its evaluation, before the jump. The key is that the checks enabled with *Discrepancy Analysis* are on the assigned values. In this sense, it is actually possible to measure icov on different sets of checks. For the evaluation of *Discrepancy Analysis*, it is interesting to measure icov either only on the directly checked instructions (the assignment statements), only on the indirectly checked instructions (the control flow statements), or all together as an aggregate.

Figures 12.3 and 12.4 report data on the icov for the benchmarks in the *CH-Stone* suite with different level of compiler optimizations. Figure 12.3 refers to optimization level -O0, while Figure 12.4 refers to optimization level -O3. The name of each benchmark is reported below every group of bars. For every benchmark there is a group of three bars.

The one on the left, composed of two parts labeled op-src and op-tmp, reports the icov for directly checked operations, i.e. assignments. The total height of the bar composed of op-src and op-tmp represents the icov for directly checked operations. The op-src part shows the portion of the covered assignments that directly assign to a variable that was actually present in the original source code. The op-tmp part, instead, represents the assignments to temporary variables, i.e. intermediate results of compound statements or variables inserted by the compiler for optimizations. These variable do not really have a counterpart in the original source code.

The second bar, in the middle of every group of three, is composed of two parts labeled cf-src and cf-tmp, and it reports the icov for indirectly checked operations, i.e. control flow operations (branches, function calls, return, statements). The total height of the bar composed of cf-src and cf-tmp represents the icov for indirectly checked operations. A branch is considered indirectly covered if the condition is directly covered, a function call is considered indirectly covered if all

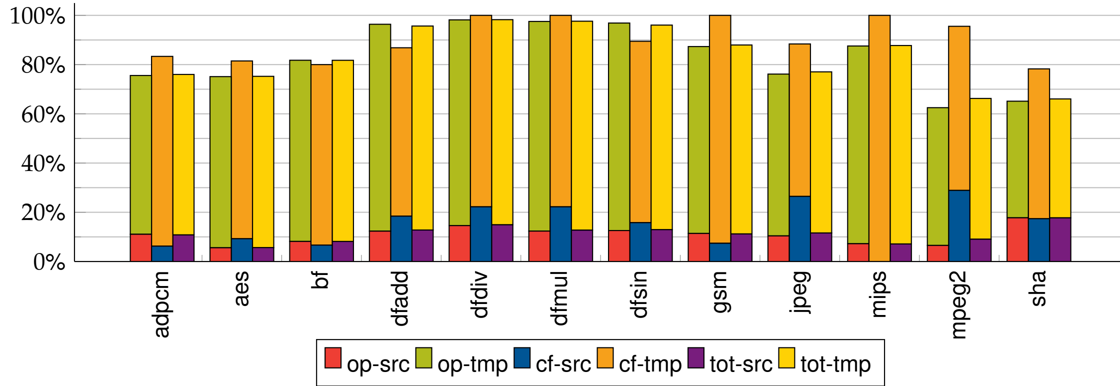


Figure 12.3: Coverage: icov for the *CHStone* benchmarks with optimization level -O0.

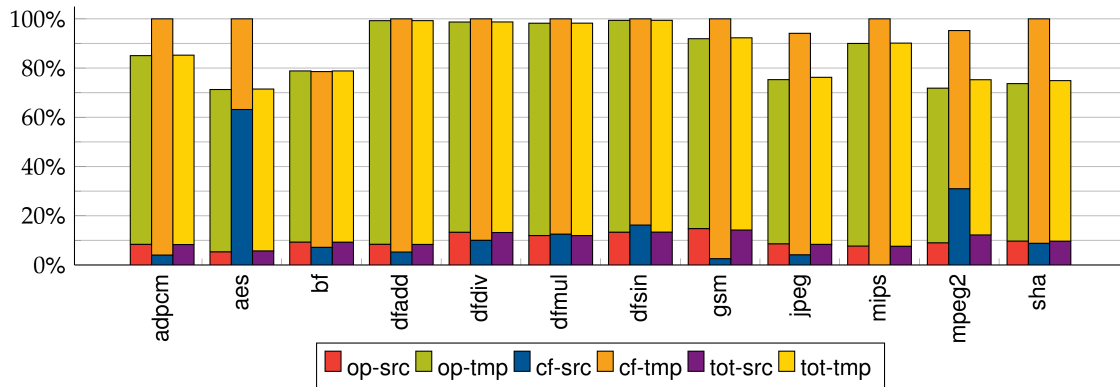


Figure 12.4: Coverage: icov for the *CHStone* benchmarks with optimization level -O3.

the arguments are covered, and a return statement is considered indirectly covered if the returned value is directly covered. The cf-src part shows the covered control flow operations whose conditions or arguments can be directly mapped onto variables in the original source code. On the other hand, the cf-tmp part is composed of the control flow operations whose conditions or arguments are temporary variables.

Bear in mind that for these control flow operations, *Discrepancy Analysis* at the control flow level provides additional checks. The static branch coverage of the *Discrepancy Analysis* presented here is always full by definition, meaning that all the branches are instrumented and control flow discrepancies are always detected by construction. For this reason, the static branch coverage is not discussed further in the thesis.

Finally, the third bar on the right in each group is composed of two parts labeled tot-src and TOT-TMP. The total height of this bar represents the aggregated icov for both directly and indirectly checked operations, i.e. for assignments and control flow operations respectively. Again, the tot-src portion represents the statements that are directly backtraceable to the original high-level source code, while the tot-tmp represents those that are not.

The first thing that appears evident from the bar plots in Figures 12.3 and 12.4 is that the percentage of checks involving temporaries is very high. With optimization level -O0 the weight of operations involving variables that were already present in the original source code (tot-src) is always less than 20%, often less than 10%. This percentage is only slightly affected by the optimization levels,

as shown in Figure 12.4. In some cases, the percentage grows, like gsm, while in other cases it drops, like sha, but always of just a few points. The reason is that optimizations affect temporaries in two different ways. On one hand, they introduce more temporary variables, because the act on the compiler's IR to optimize it. This reduces the overall impact of variables directly traceable back to high-level code. On the other hand, rising the optimization level of the compilers enables the tools to remove more useless intermediate results, to propagate constants, and to remove dead code. Given that intermediate results are not directly traceable to high-level variables, removing them increases the impact of variables directly traceable to high-level code. These two opposite effects result in the fact that the optimization level used for compilation does not significantly affect the percentage of checked operations that are related to temporary variables. As Figures 12.3 and 12.4 clearly show, most of the checks are performed on statements that involve temporary variables. This clearly shows the potential of *Discrepancy Analysis* compared to other approaches to source-level bug detection for High-Level Synthesis, which are not able to check compiler temporary variables or that force to disable compiler optimizations.

For what concerns indirectly checked operations, the percentage of checks that involve source-level variables (cf-src) is a little bit higher, up to almost 30% for jpeg-O0, mpeg2-O0, and mpeg2-O3, with a huge peak for aes-03. This peak is caused by the fact that with higher optimization levels, the compiler is able to remove a very large number of control flow operations, thanks to constant propagation and other transformations on the IR. In this way, in what remains after the optimizations, the temporary variables involved in indirectly checked operations are just a few.

On the other hand, the percentages for op-src are roughly the same to tot-src in all the benchmarks and across all the optimization levels. The reason is that directly checked operations are way more numerous than indirectly checked operations, meaning that their impact on the aggregated results is predominant.

Consider now coverage. Figures 12.3 and 12.4 show that it is always over 60%, independently of the optimization levels. This holds for directly checked operations, for indirectly checked operations, and as a consequence also for the aggregated results. The main reason why the coverage is not full involves C pointers and address arithmetic. Indeed, pointers represent a challenge for *Discrepancy Analysis*, because there is no straightforward relationship between the software address space of the high-level source code and the different possibly memory architectures of the hardware generated with High-Level Synthesis. This problem has been solved with *Address Discrepancy Analysis*, and the results concerning it are reported in Section 12.3.

Remarkably, in most cases the coverage is barely affected by optimization levels. This means that unlike many other bug detection methods, *Discrepancy Analysis* does not lose its accuracy and effectiveness when optimizations are active. This is very valuable because it allows performing bug detection on optimized designs as they are, without requiring modification to observe bugs. These modifications could, in theory, change the design up to a point where the bug is not reproducible anymore. Moreover, in all the cases, the *icov* shown in Figures 12.3 and 12.4 increases with higher optimization levels. This is a nice side-effect of the fact that more aggressive optimization remove more nodes from the IR, of-

ten substituting pointers and memory accesses with integer arithmetic and scalar values. This reduces the number of pointer operations, increasing the coverage.

It is interesting to notice that in all the cases except bf-O0, dfadd-O0, dfsin-O0, and bf-O0, the icov on the indirectly checked statements is higher than on the directly checked statements. This comes from the fact that it is less common in the evaluated benchmarks to have branch conditions or control flow statements that operate on addresses or pointers, and that pointers are the main cause of missed coverage.

Another interesting data is that it is way more common to have full coverage on indirectly checked statements. This is a consequence of the fact that in general for the considered benchmarks the indirectly checked statements are better covered than the directly checked statements. This property is even more visible in Figure 12.4, where all the benchmarks but bf, jpeg, and mpeg2 have icov = 100% for indirectly checked statements. However, this property of icov for indirectly checked statements is a direct consequence of the use of addresses and pointers in the *CHStone* benchmarks and may not hold for other cases.

There are still some cases not covered by the baseline approach, namely checks on values resulting from pointer arithmetic. This is due to the different address spaces on the host machine of the software and on the synthesized hardware. As a result, for memory-intensive benchmarks, the coverage is generally worse than others, because of the higher number of arithmetic operations between pointers. Section 12.3 discusses how they are handled by *Address Discrepancy Analysis*.

12.1.3 Other Advantages

Discrepancy Analysis has a few others interesting effects on the debugging experience, that can be observed in the following figures. Again, the reported data were collected on the *CHStone* benchmarks, and the name of each benchmark is reported below every bar in the plots, along with the optimization level used to obtain the results.

Figure 12.5 shows the percentage of signals selected in the design with our approach. It is evidently very low. It also represents the number of signals needed to ensure HW/SW execution equivalence using *Discrepancy Analysis*. Without automated signal selection, the user would typically need to find out the signals himself and to reconstruct the relationship with the original high-level source code. This would take a large amount of time especially if the user is not aware of the internal signal naming conventions of the HLS tool. This means that the automatic signal selection alone is already a huge advantage for HW designers.

But there is more. The signal selection automatically identifies in the design generated with HLS all the signals necessary for *Discrepancy Analysis*. Then there is no need to dump all the signals in the design to the VCD file. Only the interesting signals can be dumped, reducing the size of the generated waveform files. Figure 12.6 shows how much the size of the generated VCD files can be reduced by the signal selection. In some cases, this means bringing the size of the files from some GBs down to some MBs. This is also beneficial for the simulation time, which is always lower with signal selection since the I/O bottleneck to print the VCD is less significant. As a consequence, *Discrepancy Analysis* with automated signal selection significantly improves the manageability of the debugging oper-

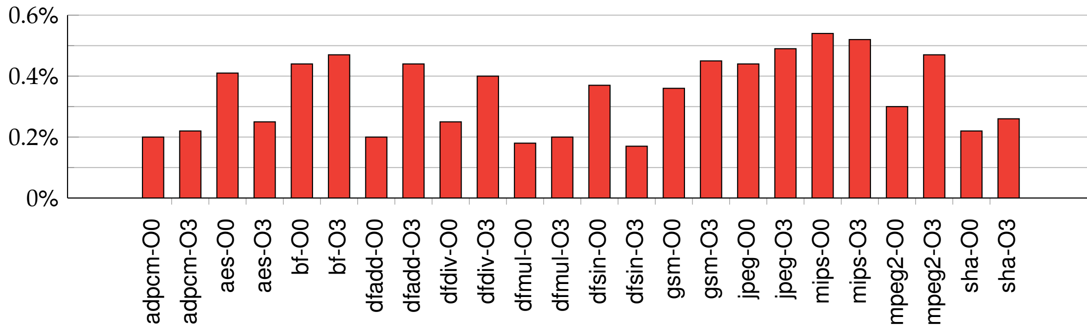


Figure 12.5: Percentage of signals selected in the design with Baseline Discrepancy Analysis.

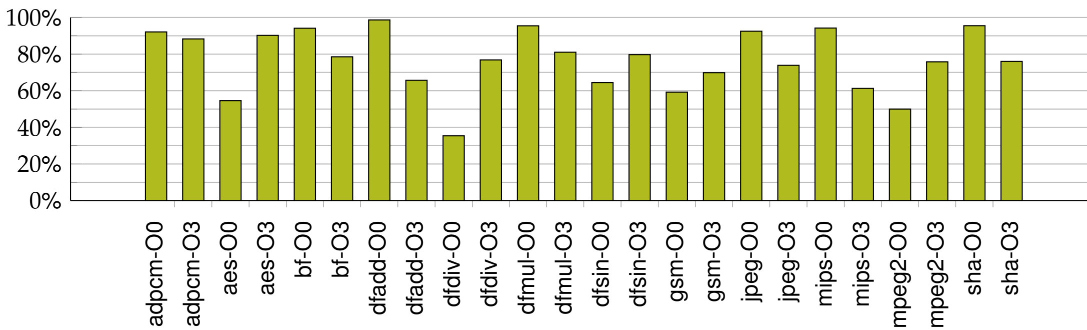


Figure 12.6: Reduction of the VCD file size using signal selection.

ations, or it can even allow debugging complex designs where the generation of the waveforms would be too expensive time-consuming to be actually feasible.

Looking at the data more closely it is possible to make some other considerations. Figure 12.5 shows that, except for a few benchmarks (aes, dfsin, and mips), the automated signals selection identifies higher percentages of significant signals in optimized design (optimization level -O3). This is reasonable, and it reflects the fact that with higher optimizations levels the HLS tool is more aggressive in removing the redundant and useless parts of the design. In this way, in the final design, more signals on the total are important for bug detection. At the same time, optimizations can considerably reduce the number of intermediate calculations necessary to compute a result, hence removing lots of intermediate results that *Discrepancy Analysis* would need to check. This is the main reason why the percentage of selected signals drops with higher optimization level in aes, dfsin, and mips.

It is also interesting to compare Figure 12.5 and Figure 12.6. At first sight, it may seem surprising not to notice a strict correlation between small numbers of selected signals and high reductions of VCD size. Indeed, there are cases (dfdiv, gsm, and mpeg2) where optimization level -O3 yields a better reduction of VCD size even if the percentage of selected signals is higher. This situation can be explained by the fact that the size of the VCD file is not only affected by the number of traced signals but also from the activity of those signals during the whole simulations. Tracing a signal with many changes during the execution requires much more space compared to a signal that only changes a few times. For this reason, it is not straightforward to correlate the number and the percentage of the selected signals with the reduction of the VCD size.

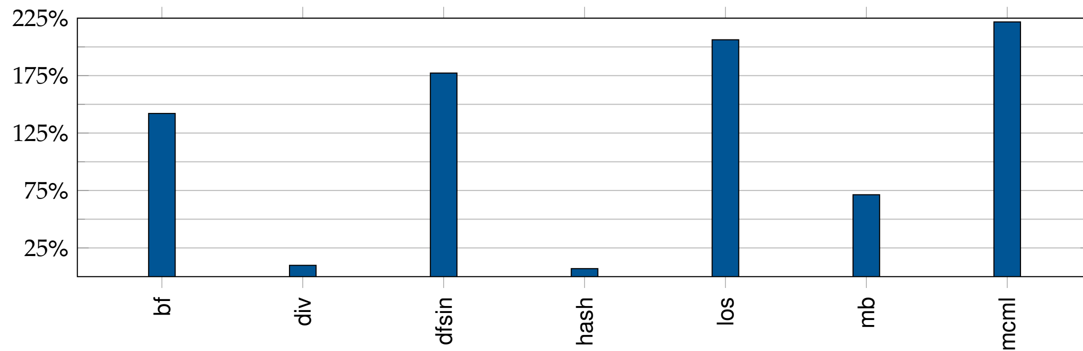


Figure 12.7: Time overhead of *Discrepancy Analysis*, compared to simulation.

12.2 Multi-Threaded

This section discusses the results obtained with *Discrepancy Analysis* on multi-threaded programs. The set of benchmarks is different from the *CHStone* suite used for serial code. Instead, this scenario was evaluated using 7 benchmarks that have already been used to evaluate HLS of OpenMP programs [17]. Coverage is not discussed here because there are no significant variations from the serial code. Instead, Section 12.2.1 discusses the performance and the scalability of the bug detection for multi-threaded programs. Then, Section 12.2.2 analyzes how the signal selection is still able to reduce runtimes and waveform sizes even with the increased complexity due to multi-threading. Finally, Section 12.2.3 describes the main limitations of the support for multi-threaded code, pointing out the possible directions of future research.

12.2.1 Performance

The performance of multi-threaded Discrepancy Analysis was evaluated measuring its execution time when the generated design was free of bugs. 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 mismatch. Without bugs, the algorithm for automated bug detection 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 10.2.2. 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 12.7. 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

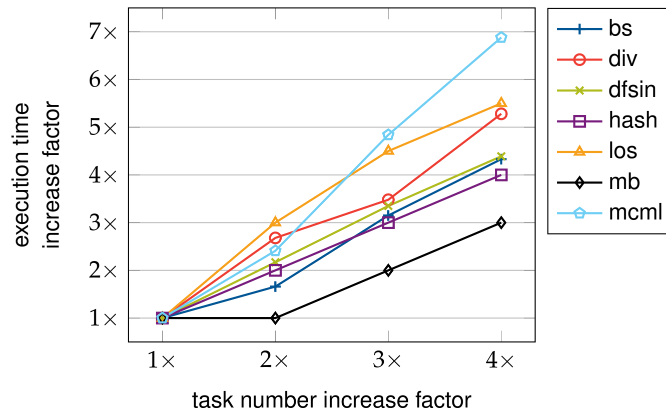


Figure 12.8: Correlation between length of execution traces and execution time of *Discrepancy Analysis*.

sharing this number is limited. With a larger number of traces the work that the bug detection algorithm 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 Section 8. 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 12.8 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.

12.2.2 Other Advantages

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 to inspect them manually. This often leads to waveform files of unmanageable size, especially with multi-threaded hardware designs, where more spatial and logic parallelism is exploited. With Discrepancy Analysis only the necessary signals are actually printed, decreasing the impact of the I/O operations on the simulation.

Figure 12.9 reports two datasets:

- the reduction of the size of the VCD files with Discrepancy Analysis;

- 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. Clearly, 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 not always evident, like 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 operations. These fixed costs are more significant on smaller benchmarks and cannot be avoided with the 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.

12.2.3 False Positives and Other Limitations

The major limitation of the approach is that the algorithm for debugging multi-threaded code described in Section 7.3 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. In order 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 7.3. The same holds in presence of synchronization directives, like locks and critical sections, with non-deterministic outcomes, such as a shared

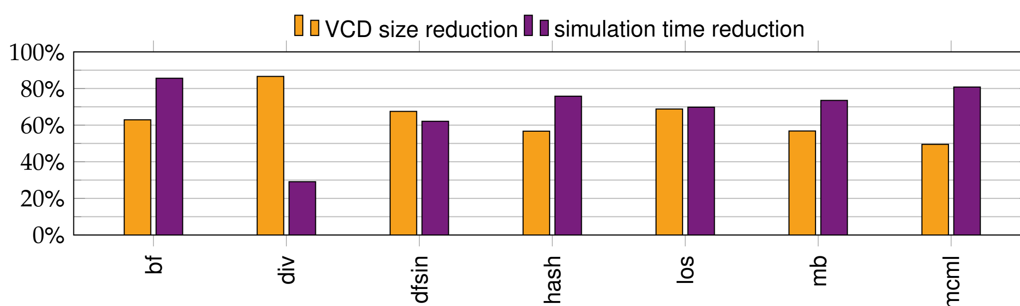


Figure 12.9: Reduction of VCD size and simulation time when Discrepancy Analysis is enabled.

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 7.4(a), or with user-defined reduction. For more complex use cases this may not be entirely possible and is definitely worth further investigation.

12.3 Address Discrepancy Analysis

This section reports the results obtained for the evaluation of the *Address Discrepancy Analysis* described in Chapter 8.

In particular, Section 12.3.1 describes the significance of address operations on the benchmarks used in the rest of the section. This is important to give a rough estimate of the amount of work that *Address Discrepancy Analysis* has to perform for each example and to shed some light on the other results.

Section 12.3.2 reports report results about alias analysis. This is useful because *Address Discrepancy Analysis* partially relies on alias analysis results, and enables to reason about the other results.

Then, Section 12.3.3 discusses the performance of the extended algorithm for automated bug detection on pointers and memory accesses introduced in Section 8.2, compared to the baseline and to simulation.

Finally, Section 12.3.4 focuses on the coverage, using the icov coverage metric previously introduced in Section 12.1.2, and showing how *Address Discrepancy Analysis* allows to significantly improve the coverage compared to the baseline reported in Section 12.1.2.

All the results presented in the remainder of this section have been collected with BAMBU on both *CHStone* and GCC *C-torture* tests. For the *CHStone* the benchmark names are reported with the used optimization flags. For GCC *C-torture* only aggregated data are reported in the results because the benchmark set is far too broad to entirely fit here.

12.3.1 Significance of Pointers and Address Operations

The first interesting information collected from the experiments concerns the importance of pointer operations on the selected benchmarks. This is important for two reasons. First, it shows that the problem tackled by *Address Discrepancy Analysis* is worth solving because it is common for pointer operations to have a significant weight in hardware generated with High-Level Synthesis. Second, these data are very useful to give a clearer interpretation of the results on the performance and on the coverage, reported in Sections 12.3.3 and 12.3.4 respectively.

Figure 12.10 reports the percentage of the operations that involve pointers and addresses, on the total operations executed *at runtime*. It is important to stress that, unlike the coverage results, this measurement is on *runtime* data. The reason is that the focus here is not on measuring how well *Address Discrepancy Analysis* can check all the possible operations, but actually what is the real amount of work that it has to execute to run the automated bug detection algorithm. This amount

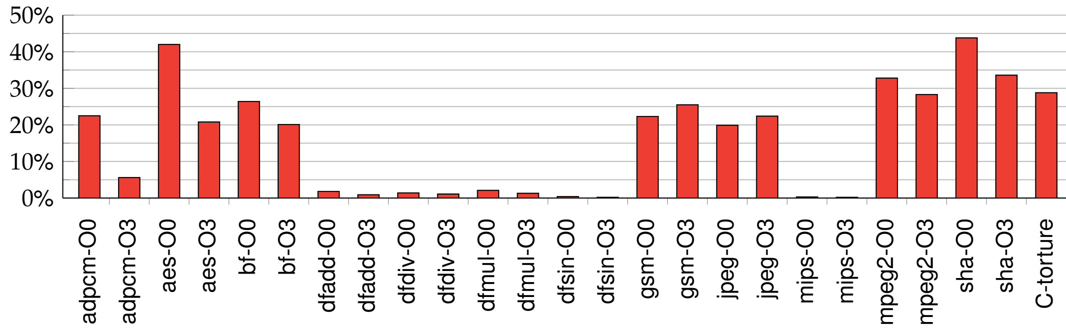


Figure 12.10: Percentage of pointer operations on the total operations executed at runtime.

of work is not related to coverage, but it is related to the actual number of pointer operations executed at runtime, independently of their absolute weight on the total number of statements measured statically. What is important here is how many addresses do actually need to be handled by *Address Discrepancy Analysis*, because this is related both to the performance of the algorithm and to the amount of work that a human designer would have to do to solve the same problem.

Figure 12.10 shows that the impact of pointer operations for memory-intensive applications is often higher than 20%, and it can even be higher than 40%. Moreover, on the C-torture tests, the impact of pointer operations is about 28%. This shows that it is perfectly possible that applications designed for High-Level Synthesis, like mpeg2 or sha in the *CHStone*, have higher amounts of pointer operations than tests designed to stress compilers on pointers. This means that *Address Discrepancy Analysis* is solving a non-trivial problem, especially in large designs.

Without *Address Discrepancy Analysis*, the developers would need to reconstruct the address translation map manually. This takes a long time and it requires memory allocation information from the High-Level Synthesis tool, with details of the addresses, the memory alignment and what are the bit patterns used as addresses in hardware. Moreover, the software memory map with stack-allocated data should be retrieved. Finally, the Algorithm 3 would have to be executed manually. In particular, the user would need to decode manually the values of the hardware signals to retrieve the memory locations, which is an operation that requires a good understanding of the internal memory allocation used in High-Level Synthesis. These operations would need to be performed manually for each and every one of the pointers manipulated during the execution, to actually identify and isolate the first bug.

With *Address Discrepancy Analysis*, instead, all the process can be automated. This effectively removes user interaction, avoiding the long, burdensome and error-prone tasks of bug isolation, and leaving to the designers only the task to figure out what introduced the bug identified automatically. This significantly improves the debugging experience, cuts time and costs, and is suitable for use in automated regression testing.

12.3.2 Alias Analysis Results

Figure 12.11 reports the percentage of pointers in the C code for which the alias analysis is *fully resolved* at compile time. Alias analysis for a given pointer is *fully resolved* if the compiler can prove at compile time that the points-to set for

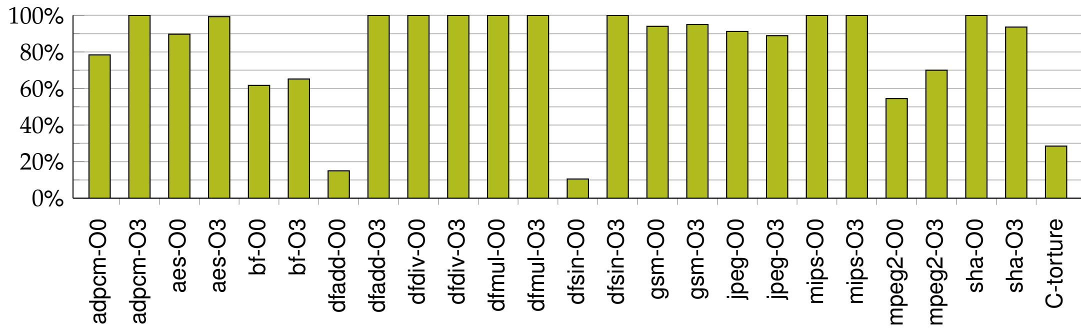


Figure 12.11: Percentage of pointers for which alias analysis is fully resolved.

that pointer is finite. This means that the compiler knows with certainty all the possible locations that can be pointed to by a pointer during the entire execution of a program, for all the possible inputs provided to it. For pointers whose alias analysis is *fully resolved Discrepancy Analysis* is more effective. If a bug affects one of them, the Algorithm 3 is always able to find it by design.

Figure 12.11 shows that for the benchmarks with the most elementary pointer manipulations (dfadd, dfdiv, dmful, and mips) the alias analysis is always fully resolved independently of the optimization levels. For others, like adpcm and dfsin, higher optimization levels actually increase the fully resolved pointers. The reason is that more aggressive compiler optimizations can effectively remove wild pointer operations, leaving intact only the fully resolved ones. However, the opposite can also happen, like for sha, where optimization level -O3 decreases the fully resolved pointers. The reason is that one of the optimizations transforms some pointer arithmetic into more efficient integer operations, thus confusing the alias analysis.

When the analysis is *not fully resolved*, *Address Discrepancy Analysis* may have to give up in some cases, without being able to decide if there is a mismatch software and hardware. If the analyzed pointer is not in range for any memory-allocated variable (a so called wild pointer), Algorithm 3 never returns a mismatch (lines 14-16). The reason is that only in-range memory locations are mapped in hardware, thus out-of-bound addresses cannot be checked. ASAN is used to ensure that there are no wild pointer dereferences, but it is interesting to measure how many times *Address Discrepancy Analysis* actually gives up on a comparison. The experiments showed that this never happens for the *CHStone* tests. This means that even if the alias analysis is non-trivial the Address Space Translation Scheme (ASTS) is able to check all the addresses. Interestingly the percentage of give up is very small (0.004%) also for the GCC *C-torture*, even if they are precisely designed to stress-test the compiler on pointer arithmetic. The good results on the *CHStone* benchmarks show that the approach is perfectly suitable for real use cases. On the other hand, the fact that even on the GCC *C-torture* test there are so few give ups confirms that the ASTS can handle a large variety of situations including several corner cases of allowed operations with C pointers.

12.3.3 Performance

Knowing the importance of pointer operations on the analyzed benchmarks, it is now possible to have more insights on the performances of *Address Discrepancy*

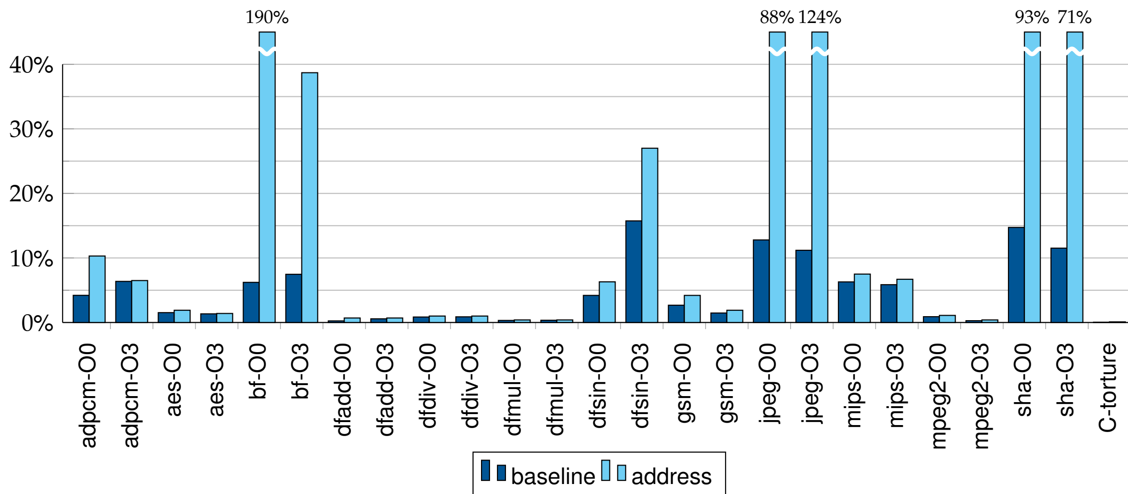


Figure 12.12: Time overhead of *Address Discrepancy Analysis*, and of the baseline *Discrepancy Analysis*. Both the overheads are intended compared to the simulation time.

Analysis. Again, the performance of *Address Discrepancy Analysis* is measured in terms of execution time overhead compared to simulation, that was executed with ModelSim SE-64 10.5. This overhead is reported in Figure 12.12. The darker bars, labeled baseline, are the overhead of the baseline version of *Discrepancy Analysis*, as already reported in Figure 12.1. The lighter bars, labeled address, are the overhead of the full *Address Discrepancy Analysis*. The higher bars have been trimmed to avoid losing visibility on the smaller results, but their actual value is reported explicitly close to the top of each bar.

As expected, the time overhead for *Address Discrepancy Analysis* (address in the figure) is always higher than the overhead for the baseline version measured in Section 12.3.3 (baseline in the figure).

When the number of address operations is negligible (dfadd, dfmul, dfdiv, dfsin, mips) the execution time overhead is not significant. This is reasonable because *Discrepancy Analysis* is very fast on integers and floating points, since there are no ASTS lookups. For the GCC *C-torture* tests, the overhead is also negligible because they are really small programs.

On the other hand, when the address operations are more than about 20%, it is harder to find a straightforward relationship between the coverage metrics and the performance overhead. The main reason is to be attributed to the results of the alias analysis. Indeed, even if the alias analysis is fully resolved it may be possible that a single pointer variable points to different memory locations (if the location set for that pointer has more than one element). Things are even worse if the alias analysis is not resolved.

One example is represented by adpcm. Pointer operations constitute 22.5% of the operations in adpcm-O0, but thanks to optimizations they are reduced to 5.6% for adpcm-O3. At the same time, for adpcm-O0 alias analysis is fully resolved for 78.4% of pointers, whereas for adpcm-O0 alias analysis is fully resolved for 100% of pointers. As a consequence, Figure 12.12 shows that the performance overhead for *Address Discrepancy Analysis* on adpcm-O3 is almost the same as the baseline, while it is more than double for adpcm-O0.

This kind of interactions explain the difficulties in finding an exact analytical

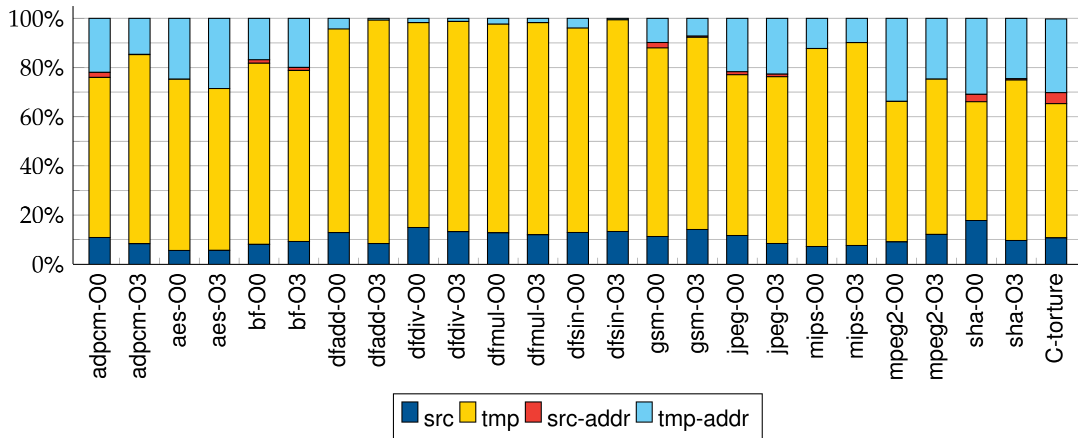


Figure 12.13: Coverage: icov of *Address Discrepancy Analysis* for the *CHStone* and *C-torture* benchmarks.

model to estimate beforehand the execution time overhead of *Address Discrepancy Analysis*. Indeed, the combined effect of alias analysis and of the percentage of pointer operations in a program can have a big impact on the execution time of Algorithm 3, because several lookups in the ASTS may be required. However, from the experiments turned out that even in the worst cases the overhead was less than 200%, which is not a huge overhead considering the complexity of *Address Discrepancy Analysis* and the time saved avoiding user interaction.

12.3.4 Coverage

The most encouraging results about *Address Discrepancy Analysis* are related to coverage. Figure 12.13 reports the coverage results measured with the icov metric introduced in Section 12.1.2. Also in this figure, the bars have been divided into multiple portions to better discuss the results. Portion src includes the checked operations that do not involve pointers and that are directly traceable to variables in the original high-level source code. Portion tmp includes the checked operations that involve compiler temporary variables but not pointers and memory accesses. Portion src-addr includes checked operations that involve pointers and memory accesses and that are traceable to variables present in the original high-level source code. Finally, portion tmp-addr includes checked operations involving pointers and memory accesses and temporary variables.

The first thing to notice is that with the introduction of *Address Discrepancy Analysis* it is possible to achieve full coverage on all the *CHStone* benchmarks. Moreover, the icov is very high (99%) also for the C-torture, which is even more important, given that these tests have been designed to stress test compilers on the quirks of C pointers.

Another important observation is that the larger part of all the checked operations (both involving pointers and not) involves compiler temporary variables. The complete visibility on temporaries still represents one of the main advantages of *Discrepancy Analysis* compared to other debugging methodologies for High-Level Synthesis, even if some other research efforts are headed in the same direction [27].

Summary

This chapter provided a broad and deep overview of the results gathered during the evaluation *Simulation-Based Offline Discrepancy Analysis*, demonstrating the improvements it can bring to the debugging experience for circuits generated with High-Level Synthesis. In particular, Section 12.1 was focused on the baseline implementation of *Discrepancy Analysis*, still without considering pointers and multithreaded programs. Then, Section 12.2 enlarges the scope to multithreaded code, showing the scalability of the approach. Finally, Section 12.3 showed how *Address Discrepancy Analysis* can effectively overcome most of the biggest limitations of the baseline implementation, concluding the discussion on *Simulation-Based Offline Discrepancy Analysis*.

Chapter 13 will now describe the results obtained on the *On-Chip Online Discrepancy Analysis*, closing the discussion on the results. Chapter 14 then summarizes the main advantages and limitations emerged by the analysis of the results, along with some possible future direction of research.

13

On-Chip Discrepancy Analysis

This chapter discusses the results obtained during the evaluation of the *online on-chip Discrepancy Analysis* described in Chapter 9. To evaluate it, the implementation has been integrated into BAMBU and will be released with a future version of *PandA* [69], an open source publicly available framework for High-Level Synthesis developed at Politecnico di Milano. To support it, two main changes have been applied to the framework:

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

The methodology flow has been tested on the *CHStone* benchmarks [33], a suite composed of 12 C programs, explicitly collected for representing all the possible scenarios which have to be addressed by a High-Level Synthesis tool. The benchmarks have been translated in Verilog with the default configuration of the *PandA* framework targeting a Stratix V device (5SGXEA7N2F45C1) from Intel (former Altera) with a target frequency of 200MHz, using Quartus Prime Standard Edition 17.0 for synthesis.

Section 13.1 discusses the reduction of memory footprint obtained with optimized EPP, Section 13.2 shows the overhead introduced by the checkers, and Section 13.3 outlines the limits of the approach.

13.1 Memory Usage

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

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

SoA: the list of the traversed states is compressed with the technique presented in [26].

EPP: the list of the traversed states is encoded with the Efficient Path Profiling as described in Section 9.3.

EOPT: the execution traces are encoded with EPP and then compressed with the technique described in Section 9.4.

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

Table 13.1: Memory usage, in bits, for storing the compressed execution traces on-chip.

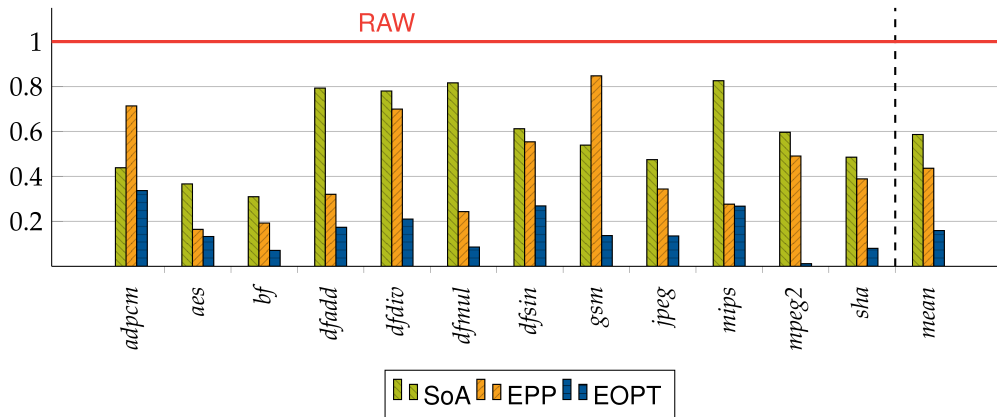


Figure 13.1: Memory usage normalized with respect to RAW.

Note that the results obtained with the SoA technique depend on the encoding of the states. The states of the FSM produced by *PandA* are binary encoded and the states are numbered according to a depth-first visit which increases the opportunities of compression of the technique used in SoA. In their work, the authors [26] identify the optimal number of metadata bits to use for compression, which is 6 for their implementation. Since the optimal value depends on the structure and the encoding of the FSM, the fixed number of bits identified in [26] have not been used here. Instead, for a fair comparison, the optimal size of the metadata was recomputed for every single function also for SoA, otherwise, the results would be too biased in favor of the approach presented here.

Table 13.1 reports the results obtained with each technique on each benchmark. Figure 13.1 reports the same data normalized with respect to the non-compressed trace. The red line marks the RAW results, which represents the baseline.

The results show that SoA compresses better the benchmarks which are characterized by data dominated computation (*aes*, *blowfish*, *jpeg*, *mpeg2*, *sha*), while it does not provide significant benefits on the control dominated benchmarks (*adpcm*, *dfadd*, *dfdiv*, *dfmul*, *dfsin*, *gsm*, *mips*). Indeed, the latter type of benchmarks has Control Flow Graphs (and so also Finite State Machines) with very large number of branches. This means that the number of transitions between states with consecutive encoding is limited, hampering the compression mechanism. On the

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

The results about EOPT show that the compression of the EPP traces described in Section 9.4 is effectively able to overcome this issue: the sizes of the compressed traces of *adpcm*, *blowfish*, *dfdiv*, *dfsin*, *gsm*, *jpeg*, *mpeg2*, and *sha* are significantly reduced with respect to EPP. However, the optimization introduced by EOPT provides benefits not only for the benchmarks characterized by loops without internal branches. If the execution trace is characterized by the consecutive repetitions of the same path inside a loop (even if it contains multiple branches), EOPT can further compress the execution trace. For this reason, a significant improvement is also noticeable in the compression of the traces for *dfmul*.

13.2 Overhead of the Tracing Logic

Table 13.2 reports the synthesis results obtained with Quartus Prime [40] after place-and-route phase for the accelerators produced by the BAMBU High-Level Synthesis flow, with and without the addition of checkers proposed in this chapter. For each benchmark, the table reports the number of checkers, the obtained maximum frequency, the area overhead in terms of Adaptive Logic Modules (ALMs) and the dynamic power consumption.

The first thing to notice is that the number of checkers, reported in the column #chk, varies across benchmarks. Given that a checker is generated for every checked FSM, the actual number corresponds to the number of functions for which BAMBU generates an FSM and that are actually executed in the reference trace. The second interesting information is that the variation of achieved clock frequency when the checkers are integrated into the design is not significant. There are cases, like *sha* and *mips*, where it decreases of about 30 MHz, but

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

Table 13.2: Synthesis results after place and route. Columns *nochk* and *chk* refer to the circuit without and with control flow checkers.

they are also the cases where the achieved frequency without checkers was much higher than the target of 200 MHz. Despite that for these benchmarks the introduction of the checkers changes the critical paths of the circuit, the newly added paths do not introduce any timing violation since there is a significant margin between their delays and the required clock period. In other cases, like *blowfish* and *gsm*, the frequency even increases of a few MHz. In all the cases the circuits with and without checkers meet the target frequency of 200 MHz.

Results about area overhead are reported in terms of ALMs, which are the basic building blocks of the Stratix FPGA. Every ALM has 8 inputs and it consists of combinational logic and four registers. It can be configured to implement various combinations of two functions with variable numbers of inputs, as described in [5]. The introduced overhead in terms of ALMs is correlated with the number of checkers instantiated in every benchmark. This can be inferred by the fact that *jpeg*, the benchmark with the highest number of checkers, also exhibits the largest ALM increase (+1051), while three out of the four benchmarks with only one checker (*dfmul*, *dfadd*, *mips*) are also those with the lowest ALM increase (+89, +76, +20, respectively). Clearly, these values have a different impact in percentage on each benchmark, because their sizes are very different, but the data are concordant with the fact that the main architecture of every checker is the same across benchmark. The only difference is represented by the number of bits necessary for the trace and the *increments memory* as described in Section 9.5. This difference is also the main cause of noise in the correlation between area and number of checkers. An example of this effect is that *adpcm*, which has only one checker, shows an ALM increase of +338, while *mpeg2*, with 4 checkers only increases of +349 ALMs.

Finally, the last two columns in Table 13.2 report the dynamic power consumption of the benchmarks, with and without checkers. The static power dissipation is not reported because it is always equal to 1515 mW $\pm$ 2 mW across all the tests, with and without checkers. The estimation is obtained with the Quartus Prime PowerPlay Power Analyzer, after place-and-route. Given that the accuracy of the tool is $\pm$ 20% compared with the results that can be obtained on silicon (ac-

cording to the documentation), it is difficult to deduce an exact equation to estimate power dissipation before synthesis. However, from the data, it is possible to see that there are three main contributions to the increase of power consumption caused by the checkers: the size of the memory used for the traces, the increased switching activity that is measured when the compression is more aggressive, and again the number of the checkers. The effect of the number of the checkers and the memory used for the traces is evident in *jpeg*, which has the highest number of checkers and the largest number of memories and shows a rise of 60.14 mW. The significance of the increased switching activity due to compression can be seen on *mpeg2*. This is the case where the EOPT shows the highest compression rate and, from the table, it is possible to see that the dynamic power consumption increases by 12.73 mW (30%) even with the insertion of a single checker. In other cases, these effects are not evident.

13.3 Limitations of the Proposed Approach

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

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

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

Summary

This chapter concludes the discussion on the results, discussing several aspects of the *Control Flow Checkers* implemented with *On-Chip Discrepancy Analysis*. The reported data include the memory consumption (Section 13.1), and the overhead of the tracing logic, in terms of area, frequency, and power consumption (Section 13.2). The results look very promising, showing that using EPP it is possible to greatly reduce the memory footprint compared to the previous state-of-the-art,

with a minimal impact on the area, the frequency, and the power consumption. However, the methodology still has some limitations, reported in Chapter 13.3, mainly related to the threading model it assumes, and to the fact that it is only focused on control flow. Chapter 14 summarizes the results and the features of the whole work, integrating suggestions on possible future directions of research to overcome these limitations.

14

Conclusion and Future Research

As High-Level Synthesis is steadily becoming more attractive for providing hardware acceleration and for energy-efficient High-Performance Computing, dedicated debugging methodologies for circuits generated with HLS are becoming a vital component of the ecosystem, essential to really achieve the promises of increasing designers' productivity and reducing time-to-market. This thesis tries to lay the foundation for an effective and accurate methodology for automated bug detection and isolation for designs generated with High-Level Synthesis: *Discrepancy Analysis*.

In particular, Chapter 5 introduces the concept of equivalence between hardware and software executions, which the rest of the work then builds upon. The clear explanation of this concept is currently what misses in all the other research contribution on automated bug detection using ideas similar to *Discrepancy Analysis*. Indeed, most of the works on automated bug detection simply rely on intuitive ideas, without clearly explaining what do they mean by equivalence between hardware and software, or without explaining the granularity of the checks they implement or the coverage of their methods.

Chapter 6 then shows how this specific model of equivalence can be used to implement two different flows for automated bug detection: one for *Offline Simulation-Based Discrepancy Analysis* and the other for *Online On-Chip Discrepancy Analysis*. This shows how versatile the concept of equivalence is, and how the methodology can be adapted to different scenarios.

Chapter 7 then discusses in detail *Offline Simulation-Based Discrepancy Analysis*, showing how it can also be extended to support High-Level Synthesis of multithreaded code. Chapter 8 explains how to resolve the complicated hardware/-software address mapping, to support automated bug detection on pointers and memory accesses. Chapter 13, instead, describes in detail *Online On-Chip Discrepancy Analysis*, concluding the description of the methodology.

Part III reports the results of a wide number of tests, designed to evaluate and measure different aspects of the methodology. Chapter 10 describes the different benchmarks and the different experimental setups used to test different features, while Chapter 11 discusses the classes of bugs that can be detected by *Discrepancy Analysis* in different scenarios. Chapter 12 discusses performance, coverage, scalability, and other advantages of *Offline Simulation-Based Discrepancy Analysis*, starting with the baseline implementation and then focusing on support for multithreaded code and `C` pointers. Chapter 13, instead, reports results on *Online On-Chip Discrepancy Analysis*, its memory footprint and the other effects it has on the generated designs in terms of area, frequency, and power consumption.

The definition of the methodology and the experimental results show that *Dis-*

crepancy Analysis is accurate and reliable. The evaluations performed in a variety of scenarios demonstrates that proposed approach successfully tackles the challenges of debugging circuits generated with High-Level Synthesis:

1. ✓ manage complexity on behalf of users;
2. ✓ help and guide users in bug detection and isolation;
3. ✓ identify relevant signals in hardware and backtrack bugs to original code;
4. ✓ handle compiler optimizations and bugs introduced by HLS;
5. ✓ handle different hardware/software memory architectures and mappings;
6. ✓ handle HLS of multithreaded code;
7. ✓ handle integration of external components.

Points 1, 2 and 3 are different aspects of the same design concept behind *Discrepancy Analysis*. If High-Level Synthesis really wants to improve designers' productivity and reduce time-to-market, it is not enough to abstract away hardware details during the design stage. Given that most of the time for producing a circuit is actually spent on testing, debugging and verification, it is necessary to have a comprehensive methodology that has access to the same information HLS has access to, so that it can abstract away the details, reduce user interaction, and really increase productivity. This is what *Discrepancy Analysis* tries to do, relying on the definition of equivalence between hardware and software execution, and using compiler notions internally to provide a consistent and user-friendly interface to users. The definition of equivalence provided in Chapter 5 was designed with this goal in mind.

The fact that *Discrepancy Analysis* has access to all the HLS compiler information also means that it is possible to achieve goals 4, 5, and 6, as described in Chapters 7 and 8.

The different classes of bugs identified in Chapter 11 demonstrate that the different flavors of *Discrepancy Analysis* are able to identify several classes of bugs in different scenarios. In particular, *Discrepancy Analysis* can effectively find bugs related to the original source code, coming from third-party libraries of hardware components (point 7), as well as subtle bugs introduced by the HLS compiler itself. These classes of bugs involve different parts of the original `C` code or of the generated circuits and are associated with different stages of the High-Level Synthesis process. All these characteristics suggest that *Discrepancy Analysis* can be successfully used in development environment based on regression testing and continuous integration, both for HLS compilers and HLS-based hardware designs. As a matter of fact, this is what is already currently done for the development of the BAMBU HLS tool used for the implementation of this work.

This obviously is not to say that there is nothing more to investigate. The experiments reported in Part III also brought to the surface many limitations of *Discrepancy Analysis*, that deserve more research.

One of the questions that were not analyzed in this work is how to generate input tests to provide good dynamic coverage of the designs, so that there is a higher chance to activate bugs that would otherwise go undetected. This is a very interesting topic, and the approach described here is actually orthogonal,

but it would be very interesting to couple the latest research results on test vectors generation with *Discrepancy Analysis*.

Another open problem for *Discrepancy Analysis* on multi-threaded code is how to support more parallel execution paradigms. The applications of HLS of multi-threaded programs are constantly increasing, and it is simply unrealistic to limit the supported parallelism to what is described in Section 7.3.

The other main limitation of *On-Chip Discrepancy Analysis* is represented by the fact that it only supports control flow checks. This restriction is due to the fact that the compression algorithm used for the traces is only applicable to Control Flow Traces, but it is not an excuse for not deepening the research on a possible compression algorithm suitable for OpTraces.

Finally, the model presented in the thesis only applies to High-Level Synthesis models that generate microarchitectures composed of a Finite State Machine and a DataPath. It has to be adapted to support more advanced models of computation, like for example Kahn Process Networks, and it may be not applicable to HLS of streaming applications.

Despite these limitations, *Discrepancy Analysis* provides consistent improvements on the overall debugging experience of circuits generated with High-Level Synthesis, and it is definitely something that could benefit any development environment based on High-Level Synthesis. Without digging into the open questions outlined above, it is already possible to see other possible directions of research, that are immediately viable using the infrastructure already present. One can think of controlling the granularity of the checks, limiting *Discrepancy Analysis* at the control flow level or at the operation level, to trade off execution times for precision, and allow to handle even very large designs. Another improvement would be to design an interoperable format for *Discrepancy Analysis*, to enable automated bug detection in complex designs composed of IP blocks coming from different vendors, and possibly integrating multiple different HLS tools. Work is ongoing to explore these scenarios, to make *Discrepancy Analysis* even more useful and accurate in a field that is still growing.

INDICES

List of Figures

2.1 Structure of a typical High-Level Synthesis flow. 11
2.2 Source code of a C function and the associated Control Flow Graph. 13
2.3 Scheduling of operations from a Basic Block to Finite State Machine. 15
2.4 Scheduling, allocation and binding in HLS. 16
5.1 Relationship between Control Flow Graph and Finite State Machine 44
5.2 Scheduling, allocation and binding in HLS. 45
5.3 Outline of a generic Discrepancy Analysis debug workflow. 48
6.1 Offline Simulation-Based *Discrepancy Analysis* debug flow. 52
6.2 Online On-Chip *Discrepancy Analysis* debug flow. 54
7.1 Finite State Automaton for the comparison of the traces. 60
7.2 Relationship between Control Flow Traces 62
7.3 Visualization of Hardware and Software Traces. 62
7.4 Example of OpenMP code and hardware generated from it with HLS. 65
7.5 Comparison of traces generated from multithreaded code. 66
8.1 Representation of the *Address Space Translation Scheme* 70
8.2 A C program causing a false positive 73
9.1 Example of source code to be synthesized, with Basic Blocks ids. . . 77
9.2 Control Flow Graph of the example in Figure 9.1. 77
9.3 Path Graph of the example in Figure 9.1. 77
9.4 Valid paths of the Control Flow Graph in Figure 9.2. 78
9.5 Two Path Graphs obtained with EPP from a CFG and an FSM. . . . 80
10.1 A C function with pointer operations not supported by *CTool*. . . . 95
11.1 Classification of the bugs detected with Discrepancy Analysis. . . . 104
12.1 Time overhead of *Discrepancy Analysis*. 107
12.2 *Statement coverage* for the *CHStone* benchmarks. 109
12.3 Coverage: icov for *CHStone* with optimization level -O0. 111
12.4 Coverage: icov for *CHStone* with optimization level -O3. 111
12.5 Percentage of signals selected with Baseline Discrepancy Analysis. 114
12.6 Reduction of the VCD file size using signal selection. 114
12.7 Time overhead of *Discrepancy Analysis*, compared to simulation. . . 115
12.8 Correlation between length of execution traces and execution time. 116
12.9 Reduction of VCD size and simulation time. 117
12.10 Percentage of pointer operations executed at runtime. 119
12.11 Percentage of pointers for which alias analysis is fully resolved. . . 120
12.12 Time overhead of *Address Discrepancy Analysis*. 121
12.13 Coverage: icov of *Address Discrepancy Analysis*. 122
13.1 Memory usage normalized with respect to RAW. 126

List of Tables

13.1 Memory required to store compressed execution traces on-chip. . . . 126
13.2 Synthesis results after place and route. 128

List of Algorithms

1 Discrepancy Analysis for OpTraces. 62
2 Pseudocode for the UPDATE function 63
3 CHECK algorithm for Address Discrepancy Analysis. 72

Bibliography

[1] ITRS: The International Technology Roadmap for Semiconductors. https://www.semiconductors.org/main/2009_international_technology_roadmap_for_semiconductors_itrs/, 2009.

[2] 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)*, pages 1–3957, Sept 2016.

[3] A. Aboagye, M. Patel, and N. Vig. Standing Up to the Semiconductor Verification Challenge. *McKinsey on Semiconductors*, Oct. 2014.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

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

[6] D. L. Andrews, R. Sass, E. K. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp. Achieving Programming Model Abstractions for Reconfigurable Computing. *IEEE Transactions on Very Large Scale Integration VLSI Systems*, 16(1):34–44, 2008.

[7] T. Ball and J. R. Larus. Efficient Path Profiling. In S. W. Melvin and S. Beaty, editors, *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 29, Paris, France, December 2-4, 1996*, Washington, DC, USA, 1996. IEEE Computer Society.

[8] Y. Ben-Asher and N. Rotem. Using Memory Profile Analysis for Automatic Synthesis of Pointers Code. *ACM Transactions on Embedded Computing Systems*, 12(3):68:1–68:21, 2013.

[9] M. Ben Hammouda, P. Coussy, and L. Lagadec. A Design Approach to Automatically Generate On-chip Monitors During High-level Synthesis of Hardware Accelerator. In *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*, GLSVLSI '14, pages 273–278, New York, NY, USA, 2014. ACM.

[10] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Softw. Pract. Exper.*, 28(8):859–881, July 1998.

[11] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, and D. Jimenez-Gonzalez. OpenMP Extensions for FPGA Accelerators. In *2009 International Symposium on Systems, Architectures, Modeling, and Simulation*, pages 17–24, July 2009.

[12] N. Calagar, S. D. Brown, and J. H. Anderson. Source-Level Debugging for FPGA High-Level Synthesis. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[13] K. Campbell, L. He, L. Yang, S. Gurumani, K. Rupnow, and D. Chen. Debugging and Verifying SoC Designs Through Effective Cross-layer Hardware-software Co-simulation. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 7:1–7:6, New York, NY, USA, 2016. ACM.

[14] K. Campbell, D. Lin, S. Mitra, and D. Chen. 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, pages 53:1–53:6, New York, NY, USA, 2015. ACM.

[15] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. S. Czajkowski, S. D. Brown, and J. H. Anderson. LegUp: An Open-Source High-Level Synthesis Tool for FPGA-based Processor/Accelerator Systems. *ACM Transactions on Embedded Computing Systems*, 13(2):24, 2013.

[16] V. G. Castellana and F. Ferrandi. An Automated Flow for the High Level Synthesis of Coarse Grained Parallel Applications. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 294–301, Dec 2013.

[17] J. Choi, S. Brown, and J. Anderson. From Software Threads to Parallel Hardware in High-Level Synthesis for FPGAs. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 270–277, Dec 2013.

[18] J. Cong and Z. Zhang. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 433–438, 2006.

[19] J. Curreri, G. Stitt, and A. D. George. High-level synthesis techniques for in-circuit assertion-based verification. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010.

[20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[21] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From OpenCL to High-Performance Hardware on FPGAS. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534, Aug 2012.

[22] P. Fezzardi, M. Castellana, and F. Ferrandi. Trace-based Automated Logical Debugging for High-Level Synthesis Generated Circuits. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 251–258, Oct 2015.

[23] P. Fezzardi and F. Ferrandi. Automated Bug Detection for Pointers and Memory Accesses in High-Level Synthesis Compilers. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016.

[24] P. Fezzardi, M. Lattuada, and F. Ferrandi. Using Efficient Path Profiling to Optimize Memory Consumption of On-Chip Debugging for High-Level Synthesis. *ACM Transactions on Embedded Computing Systems*, 16(5s):149:1–149:19, Sept. 2017.

[25] GCC Developer Community. GCC C-torture Tests. `https://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`, 2016.

[26] J. Goeders and S. J. E. Wilton. Effective FPGA Debug for High-Level Synthesis Generated Circuits. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2014.

[27] J. Goeders and S. J. E. Wilton. Using Dynamic Signal-Tracing to Debug Compiler-Optimized HLS Circuits on FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 127–134, May 2015.

[28] J. Goeders and S. J. E. Wilton. Using Round-Robin Tracepoints to debug multithreaded HLS circuits on FPGAs. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 40–47, Dec 2015.

[29] J. Goeders and S. J. E. Wilton. Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(1):83–96, Jan 2017.

[30] R. J. Halstead and W. Najjar. Compiled Multithreaded Data Paths on FPGAs for Dynamic Workloads. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, pages 3:1–3:10, Piscataway, NJ, USA, 2013. IEEE Press.

[31] M. B. Hammouda, P. Coussy, and L. Lagadec. A Design Approach to Automatically Synthesize ANSI-C Assertions During High-Level Synthesis of Hardware Accelerators. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 165–168, June 2014.

[32] M. B. Hammouda, P. Coussy, and L. Lagadec. A Unified Design Flow to Automatically Generate On-Chip Monitors During High-Level Synthesis of Hardware Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):384–397, March 2017.

[33] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. CHStone: a Benchmark Program Suite for Practical C-based High-Level Synthesis. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, pages 1192–1195, May 2008.

[34] K. S. Hemmert, J. L. Tripp, B. L. Hutchings, and P. A. Jackson. Source Level Debugger for the Sea Cucumber Synthesizing Compiler. In *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pages 228–237, April 2003.

[35] M. Hosseinabady and J. L. Nunez-Yanez. Optimised OpenCL Workgroup Synthesis for Hybrid ARM-FPGA Devices. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, Sept 2015.

[36] H. Howe. Pre- and Postsynthesis Simulation Mismatches. In *Proceedings of the 1997 IEEE International Verilog HDL Conference (IVC '97)*, IVC '97, Washington, DC, USA, Mar. 1997. IEEE Computer Society.

[37] J. Huthmann and A. Koch. Optimized High-Level Synthesis of SMT Multi-Threaded Hardware Accelerators. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 176–183, Dec 2015.

[38] IEEE. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language - Redline. *IEEE Std 1800-2009 (Revision of IEEE Std1800-2005) - Redline*, pages 1–1346, Dec 2009.

[39] IEEE. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages 1–626, Jan 2009.

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

[41] Intel FPGA. Intel FPGA SDK for OpenCL – Programming Guide. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf`, May 2017.

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

[43] Y. Iskander, C. Patterson, and S. Craven. High-Level Abstractions and Modular Debugging for FPGA Design Validation. *ACM Transactions on Reconfigurable Technology and Systems, TRETS*, 7(1):2:1–2:22, Feb. 2014.

[44] ISO/IEC. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011.

[45] ISO/IEC. *ISO/IEC 14882:2011 Information Technology — Programming Languages — C++*. International Organization for Standardization, Geneva, Switzerland, Feb. 2012.

[46] Khronos OpenCL Working Group. The OpenCL Specification – Version 2.2. `https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf`, May 2017.

[47] J. Korinth, D. d. l. Chevallerie, and A. Koch. An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 195–198, May 2015.

[48] C. Lattner and V. Adve. LLVM: a Compilation Framework for Lifelong Program Analysis Transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, March 2004.

[49] M. Lattuada and F. Ferrandi. Code Transformations Based on Speculative SDC Scheduling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, November 2-6, 2015*, pages 71–77, 2015.

[50] S. Ma, M. Huang, and D. L. Andrews. Developing Application-Specific Multiprocessor Platforms on FPGAs. In *2012 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2012, Cancun, Mexico, December 5-7, 2012*, pages 1–6. IEEE, 2012.

[51] P. Mantovani, G. D. Guglielmo, and L. P. Carloni. High-Level Synthesis of Accelerators in Embedded Scalable Platforms. In *21st Asia and South Pacific Design Automation Conference, ASP-DAC 2016, Macao, Macao, January 25-28, 2016*, pages 204–211. IEEE, 2016.

[52] K. Memarian and P. Sewell. What Is `C` in Practice? (Cerberus Survey v2): Analysis of Responses. `http://www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html`, 2016.

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

[54] D. Mills and C. E. Cummings. RTL Coding Styles That Yield Simulation and Synthesis Mismatches. In *SNUG (Synopsys Users Group) 1999 Proceedings*, 1999.

[55] M. Minutoli, V. G. Castellana, A. Tumeo, M. Lattuada, and F. Ferrandi. Efficient Synthesis of Graph Methods: A Dynamically Scheduled Architecture. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Nov 2016.

[56] M. Minutoli, V. G. Castellana, A. Tumeo, M. Lattuada, and F. Ferrandi. Enabling the High Level Synthesis of Data Analytics Accelerators. In *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–3, Oct 2016.

[57] J. S. Monson and B. Hutchings. Using Shadow Pointers to Trace `C` Pointer Values in FPGA Circuits. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Dec 2015.

[58] J. S. Monson and B. L. Hutchings. New Approaches for In-System Debug of Behaviorally-Synthesized FPGA Circuits. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6, Sept 2014.

[59] J. S. Monson and B. L. Hutchings. Using Source-Level Transformations to Improve High-Level Synthesis Debug and Validation on FPGAs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 5–8, New York, NY, USA, 2015. ACM.

[60] 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. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016.

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

[62] T. Nguyen, Y. Cheny, K. Rupnow, S. Gurumani, and D. Chen. SoC, NoC and Hierarchical Bus Implementations of Applications on FPGAs Using the FCUDA Flow. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 661–666, July 2016.

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

[64] M. Owaida, N. Bellas, C. D. Antonopoulos, K. Daloukas, and C. Antoniadis. Massively Parallel Programming Models Used as Hardware Description Languages: The OpenCL Case. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 326–333, Nov 2011.

[65] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos. Synthesis of Platform Architectures from OpenCL Programs. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 186–193, May 2011.

[66] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu. Efficient Compilation of CUDA Kernels for High-performance Computing on FPGAs. *ACM Trans. Embed. Comput. Syst.*, 13(2):25:1–25:26, Sept. 2013.

[67] C. Pilato, F. Ferrandi, and D. Sciuto. A Design Methodology to Implement Memory Accesses in High-Level Synthesis. In *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011, part of ESWeek '11 Seventh Embedded Systems Week, Taipei, Taiwan, 9-14 October, 2011*, pages 49–58, 2011.

[68] C. Pilato, P. Mantovani, G. D. Guglielmo, and L. P. Carloni. System-Level Memory Optimization for High-Level Synthesis of Component-Based SoCs. In *2014 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2014, Uttar Pradesh, India, October 12-17, 2014*, pages 18:1–18:10, 2014.

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

[70] 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. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, June 2014.

[71] A. Ribon, B. L. Gal, C. Jégo, and D. Dallet. Assertion Support in High-Level Synthesis Design Flow. In *2011 Forum on Specification & Design Languages, FDL 2011, Oldenburg, Germany, September 13-15, 2011*, pages 1–8. IEEE, 2011.

[72] B. C. Schafer. Source Code Error Detection in High-Level Synthesis Functional Verification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(1):301–312, Jan 2016.

[73] K. Selyunin, T. Nguyen, E. Bartocci, and R. Grosu. Applying Runtime Monitoring for Automotive Electronic Development. In *Proceedings of the 16th International Conference on Runtime Verification(RV) 2016, Madrid, Spain*, 2016.

[74] L. Semeria and G. De Micheli. Resolution, Optimization, and Encoding of Pointer Variables for the Behavioral Synthesis from `C`. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(2):213–233, Feb 2001.

[75] L. Semeria, K. Sato, and G. D. Micheli. Synthesis of Hardware Models in `C` with Pointers and Complex Data Structures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(6):743–756, Dec 2001.

[76] J. Seo, T. Kim, and P. R. Panda. Memory Allocation and Mapping in High-Level Synthesis - An Integrated Approach. *IEEE Trans. VLSI Syst.*, 11(5):928–938, 2003.

[77] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In G. Heiser and W. C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 309–318. USENIX Association, 2012.

[78] V. C. Sreedhar, G. R. Gao, and Y. Lee. Identifying Loops Using DJ Graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, 1996.

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

[80] B. Steensgaard. Points-to Analysis by Type Inference of Programs with Structures and Unions. In *Compiler Construction, 6th International Conference, CC'96, Linköping, Sweden, April 24-26, 1996, Proceedings*, pages 136–150, 1996.

[81] A. Takach. High-Level Synthesis: Status, Trends, and Future Directions. *IEEE Design & Test*, 33(3):116–124, June 2016.

[82] M. Tan, B. Liu, S. Dai, and Z. Zhang. Multithreaded Pipeline Synthesis for Data-Parallel Kernels. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 718–725, Nov 2014.

[83] The OpenMP Architecture Review Board. OpenMP Application Programming Interface – Version 4.5. `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`, 11 2015.

[84] F. Vahid. Procedure Cloning: a Transformation for Improved System-Level Functional Partitioning. In *European Design and Test Conference, 1997. ED TC 97. Proceedings*, pages 487–492, Mar 1997.

[85] A. Verma, H. Zhou, S. Booth, R. King, J. Coole, A. Keep, J. Marshall, and W.-C. Feng. Developing Dynamic Profiling and Debugging Support in OpenCL for FPGAs. In *2017 54nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2017.

[86] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing Modular Hardware Accelerators in `C` with ROCCC 2.0. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 127–134, May 2010.

[87] S. Wagon. *The Banach-Tarski Paradox*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1985.

[88] Y. Wang, P. Li, and J. Cong. Theory and Algorithm for Generalized Memory Partitioning in High-Level Synthesis. In *The 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '14, Monterey, CA, USA - February 26 - 28, 2014*, pages 199–208, 2014.

[89] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong. Memory Partitioning for Multidimensional Arrays in High-Level Synthesis. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 12:1–12:8, 2013.

[90] Y. Wang, J. Yan, X. Zhou, L. Wang, W. Luk, C. Peng, and J. Tong. A Partially Reconfigurable Architecture Supporting Hardware Threads. In *2012 International Conference on Field-Programmable Technology*, pages 269–276, Dec 2012.

[91] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for `C` Programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*, page 1, 1995.

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

[93] L. Yang, S. Gurumani, D. Chen, and K. Rupnow. AutoSLIDE: Automatic Source-Level Instrumentation and Debugging for HLS. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–130, May 2016.

[94] L. Yang, M. Ikram, S. Gurumani, S. Fahmy, D. Chen, and K. Rupnow. JIT Trace-Based Verification for High-Level Synthesis. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 228–231, Dec 2015.

[95] P. Zhang, M. Huang, B. Xiao, H. Huang, and J. Cong. CMOST: a system-level FPGA compilation framework. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 158:1–158:6, 2015.