Source-Level Instrumentation for In-System Debug of High-Level Synthesis Designs for FPGA

by

Jose Pablo Pinilla

B.Eng., Universidad Pontificia Bolivariana, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

September 2016

Abstract

High-Level Synthesis (HLS) has emerged as a promising technology to reduce the time and complexity that is associated with the design of digital logic circuits. HLS tools are capable of allocating resources and scheduling operations from a software-like behavioral specification. In order to maintain the productivity promised by HLS, it is important that the designer can debug the system in the context of the high-level code. Currently, software simulations offer a quick and familiar method to target logic and syntax bugs, while software/hardware co-simulations are useful for synthesis verification. However, to analyze the behaviour of the circuit as it is running, the user is forced to understand waveforms from the synthesized design.

Debugging a system as it is running requires inserting instrumentation circuitry that gathers data regarding the operation of the circuit, and a database that maps the record entries to the original high-level variables. Previous work has proposed adding this instrumentation at the Register Transfer Level (RTL) or in the high-level source code. Source-level instrumentation provides advantages in portability, transparency, and customization. However, previous work using source-level transformations has focused on the ability to expose signals for observation rather than the construction of the instrumentation itself, thereby limiting these advantages by requiring

lower-level code manipulation.

This work shows how trace buffers and related circuitry can be inserted by automatically modifying the source-level specification of the design. The transformed code can then be synthesized using the regular HLS flow to generate the instrumented hardware description. The portability of the instrumentation is shown with synthesis results for Vivado HLS and LegUp, and compiled for Xilinx and Altera devices correspondingly. Using these HLS tools, the impact on circuit size varies from 15.3% to 52.5% and the impact on circuit speed ranges from 5.8% to 30%. We also introduce a low overhead technique named Array Duplicate Minimization (ADM) to improve trace memory efficiency. ADM improves overall debug observability by removing up to 31.7% of data duplication created between the trace memory and the circuit's memory structures.

Preface

This dissertation is original, independent work by the author, J. Pinilla

The work on this thesis, under the supervision of Prof. Steve Wilton, led to the submission of the research paper titled Enhanced Source-Level Instrumentation for FPGA In-System Debug of High-Level Synthesis Designs, accepted for oral presentation at the International Conference on Field Programmable Technology 2016 (FPT16).

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Computation scaling through recent years has not seen the same return in terms of higher operational frequency and power efficiency that was obtained from shrinking transistor sizes. As processor performance plateaus, computer architectures have moved to parallel execution models. Such models either replicate processing units or use heterogeneous computing. Heterogeneous computing merges different computing architectures into one system, where each architecture targets a specific task or type of tasks.

There are two domains in which this trend is specially important. First, embedded devices, in which energy efficiency can be achieved by offloading tasks from the main processor core to custom cores that consume less power, allowing the more powerful resources to go to a standby state. Second, in High Performance Computing (HPC) or any compute-demanding application, where higher performance can only be obtained with application-specific logic, and both the sequential processing unit(s) and custom logic can execute in tandem. However, design and fabrication of Application Specific Integrated Circuits (ASICs) is not only time consuming, but also often unaffordable.

More flexible heterogeneous computing systems integrate programmable

General Purpose (GP) devices instead of specialized ASICs. This balances the customizability and performance of ASICs, while preserving the flexibility and affordability of software programmable cores. For example, Systems on Chip (SoC) can integrate two Central Processing Units (CPUs) with architectures focused for performance and for power efficiency [7], or the CPU(s) can be integrated with a GPGPU (General Purpose Graphics Processing Unit) [25, 58] which has become attractive in scientific computing and other HPC applications. The reprogrammable many-core Single-Instruction Multiple-Data (SIMD) execution model offered by GPUs can be used to solve graph and matrix based problems, among other applications. A more suitable approach for some applications is to provide finer-grained reprogrammability, to also allow Multiple-Instruction Multiple-Data (MIMD) parallelism with custom logic design. This execution model is offered by Programmable Logic Devices (PLDs), namely, Field-Programmable Gate Arrays (FPGAs).

1.1 Field Programmable Gate Arrays

The fine-grained nature of FPGAs allows them to emulate the behaviour of any digital logic circuit. Their architecture is based on Look-Up Tables (LUTs) for digital gates, memory blocks for RAMs and ROMs, registers for sequential logic, programmable I/O blocks, and programmable switch blocks for custom interconnection. More specialized blocks are also seen in state-of-the-art devices. This extensive flexibility made these devices a suitable match for prototype development, Application Specific Integrated Cir-

cuit (ASIC) emulation for design verification, and as interfacing devices for system-level design, otherwise called "glue logic". More recently, FPGA resource density has increased to such degree that complex, multi-core, multi-clock domain systems can be fully implemented in programmable logic [26]. The easily replicable homogeneity of modern FPGA architectures has lead to impressive records for transistor density in one chip [4, 64]. FPGAs also enable a faster time-to-market; this is advantageous when compared to ASIC design [65]. At the same time, FPGAs can offer a significant increase in performance compared to CPU and GPU implementations [66].

This great potential for acceleration has been successfully put into practice for multiple applications where custom functional units and pipelined execution can be designed to take advantage of existing parallelism [55, 78]. Notable applications exist in high-demand cloud computing [11, 60], where each server node is augmented with one FPGA configured with multiple custom Processing Elements (PE). Recently, Intel Corporation acquired one of the largest FPGA companies, Altera Corporation [18], while major efforts from other companies have also been seen in order to incorporate FPGAs into mainstream computing, especially for machine learning on the cloud [8, 23, 78]. However, programming FPGAs requires greater effort than programming GPUs or CPUs in order to extract optimal efficiency [10, 26, 55, 66], mainly because these designs need to be described at a lower abstraction, requiring ample knowledge of the device architecture and hardware-specific design methodologies.

Traditionally, FPGA applications are specified using Hardware Description Languages (HDLs) which require hardware expertise (e.g. VHDL or

Verilog). HDLs are used to write a *structural* or *behavioral* description of the circuit, in which low-level logic (logic functions and flip-flops) and detailed timing requirements (rising and falling edge triggering) are specified; this is defined as the Register-Transfer Level (RTL). Other alternatives such as block diagram specifications are available, but these are often tedious to read and analyze with larger and more complex designs.

1.2 High Level Synthesis

FPGA vendors and academic research have invested significant effort into providing a software-like design environment for FPGAs in the form of High-Level Synthesis (HLS) tools. This involves automatically transforming a *behavioral description* into a digital circuit design. High-Level Synthesis (HLS) has emerged as a leading technology to reduce the design time and complexity that is associated with FPGAs, and to enable software programmers to use FPGAs in such a way that their expertise can be put into practice for compute acceleration without a steep learning curve [36, 37]. In order to do this, the behavioral description language and programming flow needs to resemble that of software.

This software-like behavioral description can be done using Domain Specific Languages (DSLs) (i.e. SystemC, BSV [57]), or subsets and extensions of existing software programming languages (i.e. Java, C, C++). The preferred language for existing and recent HLS tools is C [56]. C-based tools often use either GCC or LLVM [45], which are open source C/C++ compiler frameworks. These compilers are modified to include a new *backend*. The

backend is the last stage in the compilation flow, where the Intermediate Representation (IR) or code written with generic low-level instructions, is converted to comply with the target architecture, i.e. x86, ARM, MIPS or other binary executables in the software approach. When targeted towards FPGAs or logic design, this backend translates the IR into an HDL specification.

Most software concepts can remain unchanged in the language-subset approach, such as "calling function", "jumping to instructions" and "variable pointers", except these are translated into digital logic instead of binary instructions. This is attractive to software developers and hardware developers alike for two main reasons, portability and ease of transition. Existing software can be compiled for an FPGA device, and because HLS tools use a well-known programming language, it is more likely that the developer is already familiar with it and can focus on optimization rather than the initial implementation. Along with FPGAs and HLS tools to program them, it is necessary to have a development infrastructure. This should include assistance in the code writing process, as found in multiple Integrated Development Environments (IDE) [22], but should also allow the users to analyze and debug the behaviour of their design. Specifically, debugging should be possible in the context of the abstraction level at which the logic circuit is being designed.

1.3 HLS Debug

High-level synthesis compilers are not enough. An entire ecosystem including support for debug and optimization is required. During the design process, the developers move through multiple iterations of the design. Flaws or bugs in the execution of the circuit can appear in different stages of development and can be found using different debugging techniques. Starting from the lowest level of abstraction, debugging can target electrical bugs such as manufacturing defects, device wear out, and unmet timing. Moving up the abstraction level, bugs can generally be classified into logic, arithmetic, or syntactic. Most syntax bugs can be found statically; these are code structures that do not comply with the programming language, or are mistyped expressions (variable name mismatch, wrong operator, wrong variable type, etc.). Logic and arithmetic bugs refer to operational discrepancies from the expected behaviour (incorrect statements, loop bounds, division by zero), whether these are caused by a flawed description of the design, or by a mistake in one of the synthesis (or compile) stages. As represented in Figure 1.1, an HLS design can be debugged at three different levels, i.e Software simulation, co-simulation, and in-system debug.

1.3.1 Software Simulation

Currently, HLS tools incorporate software simulations. These offer a quick and familiar method to target logic and syntax bugs following the same standards applicable to software debugging. As previously mentioned, C-based HLS tools use standard C compiler frameworks, and the resemblance

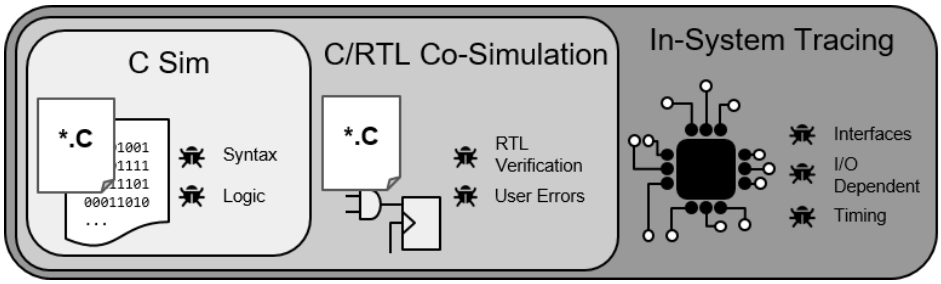


Figure 1.1: HLS Debugging Techniques

of the HLS programming flow to a software programming counterpart is such that the behavioral specification can be compiled and executed on the workstation using the regular software compilers. This type of source simulation helps finding bugs early in the design, and without the need for executing the synthesis flow.

1.3.2 Co-Simulation

Some HLS tools also provide software/hardware (C/RTL) co-simulations, which encompass the bug coverage provided by C simulations but are also useful for synthesis verification, i.e. checking for tool bugs or tool usage errors. Here, a cycle-accurate simulation of the generated circuit runs along with the binary executable. Inconsistencies can be checked during execution or by comparing return values to golden data which is useful for uncovering the root cause of errors or performance bottlenecks that arise when the code is synthesized to hardware, possibly due to incorrect compiler settings (such as *pragmas*), and to provide confidence that the HLS tool produced correct hardware.

1.3.3 In-System Debugging

Although software simulation and hardware/software co-simulation are an essential part of the HLS ecosystem, they are not sufficient to find the root cause of all bugs. Many of the most elusive bugs do not become apparent unless the design is run in-system, exercised by real input traffic, at-speed, for long periods of time. As a specific example, simulating an SoC as it boots Linux would take years in modern simulators, yet there are many types of bugs that need at least this before appearing. Further, many elusive bugs (such as those related to the timing interactions between HLS and legacy RTL blocks) may not occur unless the system is running at-speed, which in FPGA design can be accomplished from an early stage of development.

Running at-speed, however, means that data regarding the state of the circuit is being updated constantly. If exposed to the user, the amount of information regarding all available signals in the circuit would require very high throughput and I/O resources. Also, in contrast to software debugging, step by step execution of a digital logic design is often not possible for correct operation. This is due to interfaces with additional modules that can only be observed at runtime, whether because the source code is inaccessible or nonexistent at the same high level, or because the module requires external inputs from peripherals. These cases are common due to the use of external IP in most designs.

The alternative approach that is found in commercial in-system debugging solutions, is to use a trace record-replay technique. Instrumentation, or additional circuitry is added to the design in order to store the changes of

signals of interest until a predefined event is encountered. At that point in the execution, all captured data is retrieved and the behaviour of the circuit is "replayed", presented as waveforms to the designer. In Chapter 2, several examples for in-system debug are presented.

Source-Level In-System Debugging for HLS

In HLS systems, providing support for in-system debug is especially challenging, due to the mismatch between the hardware running on-chip and the HLS (software) designer's view of the system [8, 27]. A software designer views the design as a set of sequential statements with limited parallelism, while the actual hardware consists of many sequential and combinational hardware units running at the same time. A software designer does not consider the notion of a "clock" when specifying a design, yet the cycle-by-cycle behaviour is inherent in the structure and operation of the hardware. This is especially challenging if the HLS tool performs many optimizations on the code, leading to a structure and schedule that may be very unfamiliar to the designer.

Although there are many debuggers that provide visibility into the design at the system level, these tools provide information commonly presented as signal waveforms that only have meaning to a hardware designer. These signals are generated by the synthesis tool, and can be completely detached from the initial source code nomenclature and design intuition, e.g. the generated circuit will likely contain Finite State Machines (FSMs), of which the value of the register holding the current state might be of great interest for a hardware designer, however, this is meaningless from a software perspective.

To be effective and maintain the productivity promised by HLS, the in-system debug technique must present the execution in terms of C-level variables and C-level control flow, rather than presenting cycle-by-cycle waveforms that the designer must manually relate to the original C design [8]. The objective is to have *source-level in-system debug for HLS*.

This has led several research groups to develop techniques to provide a software-like view of running hardware, allowing the software designer to observe variable values and single-step code as if it was software. Early work presented a system for the JHDL-based Sea Cucumber (SC) framework [39] allowing software optimizations instead of using a debug version of the code. More recently, LegUp's release included source-level in-system debugging support [12, 27] using custom RTL instrumentation and a database to relate hardware signals to source-code statements, LLVM's Intermediate Representation (IR), and Verilog. Subsequent research [30] focused on optimizing resource utilization, successfully storing longer execution histories.

1.3.4 Instrumentation

Key to any in-system debug approach is efficient and effective instrumentation that is added to the user design to record the behaviour of the design as it runs. Since I/O pins are limited, and since it is desired to run the chip at-speed, existing systems instrument the user design by adding memories (*trace buffers*) and support circuitry to record the behaviour of key signals into these trace buffers. After the chip has been run, these trace buffers can be interrogated using a software-like debugging tool, allowing the designer to understand the behaviour of the system.

The instrumentation is built using resources of the same reconfigurable fabric used for the user's circuit. As such, this instrumentation can be added at any stage of the design process, bitstream-level [31], gate-level at compile time [2, 43], incrementally after place and route [24], RTL-level [12, 27], or in high-level source code [53]. These different approaches either need to modify the synthesis tools to insert the instrumentation while performing the synthesis process, or modify (instrument) the code before putting it through the subsequent synthesis stage.

Source-Level Instrumentation

High-level source code instrumentation, or source-level instrumentation, is an HLS specific approach in which the user's source code is *transformed* before any lower-level code generation. The motivation behind source-level instrumentation is threefold.

- First, inserting instrumentation at the source level creates a runtime-verifiable design that is portable between any HLS tool that uses the same high-level language. Source-level isntrumentation avoids the difficult task of mapping circuit-level structures to C-level elements. In solutions such as [12, 27], such a mapping requires access to a debug database from the HLS tool, making it difficult to apply the technique to a commercial HLS flow.

- Second, code transformations can be written in such a way that it takes advantage of the HLS and Logic synthesis optimizations; both the source code and the instrumentation will be optimized together.

- Third, the instrumented C code is readable and familiar to the designers, allowing them to better understand the role of the instrumentation in the debugging process.

A key challenge of instrumenting at the C level is the possibility that the overhead may quickly become overwhelming if too much instrumentation is added. In [52], it is shown experimentally that this overhead can be kept reasonable, suggesting this technique is feasible.

1.4 Contributions

In this work, a methodology is described to use *source-level instrumentation* for C-based HLS tools, to create memories and related circuitry to gather trace data that provides visibility into the operation of the circuit. This trace is then matched with a database that maps each memory entry to the high-level variables that have meaning to the designer during debug. The contributions of this work are presented here.

1. In previous work [52, 53], the focus was on connecting internal signals to Event-Observability Ports (EOPs) to provide access points in the design. These are left unconnected and meant to be connected to memories by modifying the generated RTL, or through proprietary ELAs[54]. In this work, we implement not only the connections to access points in C, but also insert trace buffer memories, as well as trace readback and related circuitry in the C-level design. This eliminates the need to make modifications after compilation, and as we will

show, can lead to further co-optimization opportunities between the trace buffer memories and the circuit memories. We experimentally evaluate the overhead associated with such instrumentation.

2. Exactly which signals or events should be instrumented is vital to the effectiveness of this technique. In this work, we distinguish between two strategies, *control flow instrumentation* and *data capture instrumentation*, each of which provides different views of the circuit's behaviour to the debugger, and evaluate the overhead for each of these strategies.

3. We introduce and evaluate a low overhead technique named Array Duplicate Minimization (ADM) to improve trace memory efficiency. ADM improves overall debug observability by removing up to 31.7% of data duplication created between the trace memory and the circuit's memory structures. This optimization is enabled by the inclusion of the trace buffer memories in the original C code.

1.5 Thesis Outline

This thesis is organized as follows. Chapter 2 presents the corresponding background about FPGA programming and debugging solutions. This is backed by references from recent surveys and applications developed using HLS that demonstrate the growing availability and usability of this type of programming environment. These examples also emphasize the need for a debugging infrastructure that allows the user to remain in the higher ab-

straction level while still taking advantage of the flexibility of the underlying hardware. Related projects on source-level debugging are also contained in Chapter 2 to establish the state of the art in instrumentation techniques.

Chapter 3 presents the debugging flow that the user can expect from using the tools created for this project and also describes the methodology followed to implement those tools. Both Control Flow and Data Capture instrumentation are presented by using code examples. Chapter 4 presents the motivation behind our Array Duplication Minimization (ADM) technique and the methodology followed for its implementation.

In Chapter 5, the quantification of the instrumented designs is presented. These results are compared with data available from related work and the feasibility, trends, and corner cases of the proposed technique are analyzed. Chapter 6 concludes and suggests future work.

Chapter 2

Related Work

This chapter presents related work, describing the state of the art in HLS in-system debugging. Initially, this chapter introduces the HLS flow and several HLS frameworks. The HLS tools chosen to evaluate the proposed approach are presented in more detail. This is followed by Section 2.2 which contains an introduction to Embedded Logic Analyzers (ELAs), a standard in-system debugging approach for RTL-based FPGA designs. Verification and debugging methods for HLS frameworks are then described in Sections 2.3 and 2.4, covering several tools and instrumentation levels. Then, in Section 2.5, source-to-source transformation, its applications, and the framework used for this purpose are reviewed.

2.1 High-Level Synthesis Frameworks

HLS tools for digital design have been a focus of research for more than three decades [70]. The idea behind HLS is to specify a behavioral description and have the Computer-Aided Design (CAD) tools find the best use of logic resources to implement the operations and variables using functional units and registers.

Recently, with the proliferation of FPGAs, a wide range of tools have

been made available by the device manufacturers, CAD tool companies, and by academic research efforts [15, 19, 44, 57, 67–69, 75, 77]. These and more are described in the latest HLS tools surveys [8, 21, 49], with the most recent one [56] listing and categorizing many of these tools. A common observation gathered from these surveys is that the quality and ease of adoption of C-based HLS tools are beyond other HLS tools using DSLs or other languages. This is measured using multiple factors, such as ease of implementation, abstraction level, supported data types, exploration, verification, area results, documentation, and learning curve.

In addition to the frameworks included in said surveys, the latest OpenCL-based products from Intel (Altera) [3] and Xilinx [74] make use of a similar infrastructure to compile C-based kernels into IP modules with the corresponding interface, to be called from a host device. Most recently, Altera announced the A++ compiler [1] for standalone IP design from C/C++ specifications. These design environments provide compelling productivity improvements for FPGA designers, and may open the field of FPGA acceleration to more designers than ever before.

2.1.1 HLS Flow

Figure 2.1 is a representation of the internal stages identified in most HLS tools. Similar to a software compilation flow, the HLS flow can be divided into three main stages: input code parsing, optimization, and unparsing. For this reason, HLS tools are commonly based on open source software-compiler infrastructures (i.e. LLVM or GCC), borrowing the nomenclature for many of their components. These three stages correspond to the frontend, opti-

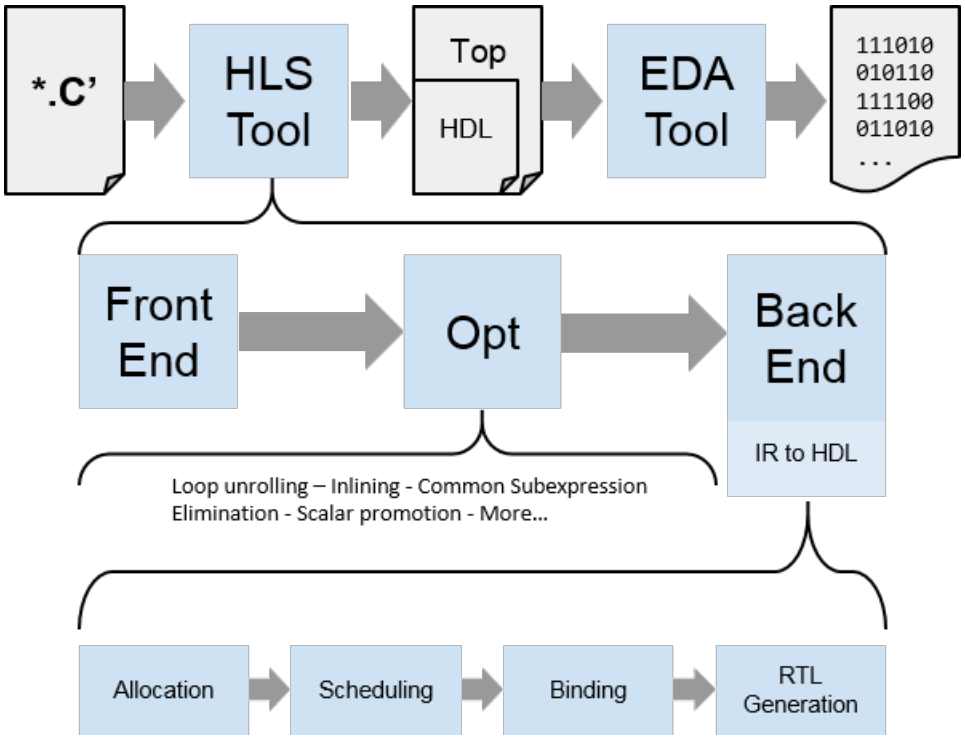


Figure 2.1: HLS Flow

mizer, and backend. An HLS flow can be seen as a software compilation, where the backend is modified in order to generate an HDL specification instead of an architecture-specific binary file. However, the frontend and optimizer are modified to target this specific flow, avoid unsupported constructs, and for optimization.

The frontend is in charge of generating a formal representation, be it an Intermediate Representation (IR) code (a generic low-level version of the user code), a Control and Data Flow Graph (CFDG), or both. The frontend can be set to recognize statements that are incompatible with hardware design, and therefore unsupported (i.e. dynamic memory allocation, system

calls, etc.). These can in turn be transformed into supported constructs, ignored (printf() statements), or synthesis flow can be halted. The optimizer, through multiple passes, transforms the IR to minimize the number of low-level instructions and, in general, the amount of resources required according to the constraints of the target architecture.

The backend consists of a set of steps that create an internal representation that can be unparsed into an HDL specification. These steps can be classified as allocation, scheduling, binding, and RTL generation.

Allocation The number and type of operations identified in the IR are mapped into functional units. All variables, registers or memory values are assigned to physical structures of the corresponding type.

Scheduling Through dependency analysis heuristics, such as the System of Difference Constraints (SDC) [14, 17], the allocated functional units are scheduled, and a set of states are defined for the execution of one or multiple threads.

Binding When possible, non conflicting functional units can be merged into one unique instance that is multiplexed throughout the execution, these will mostly apply to scarce, resource-demanding structures such as dividers and multipliers.

RTL Generation Once optimized using the least resources, the data structure used to represent the logic system architecture is transformed into an RTL representation that is possible to compile using the corresponding

EDA tool.

2.1.2 Vivado HLS

Previously owned by AutoESL under the name AutoPilot [79], this HLS tool was acquired by Xilinx in 2011 and is offered alongside the Vivado Design Suite package. The Vivado HLS IDE uses a familiar GUI, resembling that of the C/C++ Development Tool (CDT) for Eclipse [22]. As such, the IDE offers different *perspectives* or environments for Synthesis, Analysis, and Debug. Synthesis of C, C++, SystemC, and OpenCL kernels is supported, with a large set of standard circuit interfacing options and the use of TCL directives to control code optimizations such as unrolling, pipelining, inlining, chaining, memory partitioning, etc. The compile flow uses LLVM for its frontend and optimization passes. Visualization of the generated schedule of execution is possible using a Gantt Diagram representation, as well as a *raw* data report for more advanced analysis.

The Debug perspective is a C/C++ Software Debugging tool, therefore, it features a complete set of software debugging capabilities, such as breakpoint insertion, variable monitoring, custom expression analysis, and register monitoring. The Debug perspective is activated when using the C simulation tools. This does not take into account any of the synthesis results and, instead, executes a sequential version compiled for the host architecture. Therefore, as mentioned before, bugs created during the synthesis flow or activated during in-system interfaces will not be captured.

For synthesis verification, Vivado HLS offers the C/RTL Co-simulation. This is an RTL simulation using one of the provided hardware simulators

(i.e. Vivado Simulator, Modelsim, etc.) and requiring a *testbench* as a wrapper of the synthesizable code. The testbench can contain a simple return value comparison or contain complex user-designed Vector Based verification procedures. The testbench return value is the measure of success, and the tool reports this and the Co-simulation latency, or number of cycles of execution observed in the simulation.

In order to perform in-system debugging of the generated hardware, one must first export the generated RTL to a Vivado RTL project. Once synthesized, before implementation (optimization, place, route, and bitstream generation), the in-system debug flow requires the insertion of the proprietary Debug Cores provided by Xilinx, configuration of the cores' properties, and connection of the cores' probe ports to signals of interest, either as data, triggers, or both. There are no HLS related configurations or features at this design level.

2.1.3 LegUp

The LegUp framework, being developed at the University of Toronto, is an open-source HLS research tool [15]. LegUp takes ANSI-C as an input and is currently capable of generating Verilog circuits for a small set of Altera and Xilinx devices. The framework lacks a GUI interface for code editing and project management, but offers a considerable set of features configurable through TCL scripts for code optimizations and options for hardware generation, such as loop pipelining, loop unrolling, function inlining, and RAM grouping. A LegUp design choice is to group global arrays and arrays referenced in multiple functions into one memory with a single memory controller

interface. This is contrary to Vivado HLS, in which these are assigned to independent memory blocks.

The LegUp design flow offers three options for design compilation: a hardware-only implementation, a software-only alternative including a soft MIPS processor in the FPGA to execute the software provided, or a *hybrid* version that runs top-level functions in the soft processor and *calls* hardware accelerators for user-chosen functions. A profiler is also provided to inform the user regarding the behaviour of the program on the soft processor and aid in the choice of functions for acceleration.

LegUp offers an integrated in-system debugging experience. In version 4.0, LegUp provides the HLS Debugger infrastructure, which is part of the open source repository. With this tool, users can recompile the code and include debugging instrumentation. Section 2.4.2 provides a more detailed description of this infrastructure.

In order to fully support the approach used in this work, LegUp was modified to provide port declaration in the same way as is done in Vivado HLS. Appendix A explains the modification.

2.1.4 Benchmarks

The CHStone benchmark suite is widely used to evaluate HLS tools. This benchmark suite is based on a selection and creation of programs that span multiple domains with various quantifiable characteristics [38].

Although C-based HLS tools take C-like programs as input, such tools do not completely comply with the ANSI/ISO C standard and only support a subset of the C language. This benchmark suite is designed according

to these limitations by avoiding, for instance, the use of dynamic memory allocation and recursion. This, however, does not guarantee that all programs can be synthesized on every HLS tool since every tool has specific requirements for the input code. For the HLS tools used in the experiments for this work, 3 out of 12 of the original programs did not compile in at least one of the HLS tools; these have been removed from the results. The remaining majority of programs give a good representation of the behaviour of the tools with and without the debugging techniques presented herein.

2.2 FPGA Debug

Debugging a digital circuit on FPGA requires a thorough understanding of the implementation. In particular, understanding the behaviour of a design often requires observing these signals over time. Due to resource constraints, it is not possible to observe the behaviour of *all* signals, therefore, it is important for the designer to be able to select and analyze only the most relevant signals of the design [42]. Once identified, the behaviour of those signals of interest can be exposed to the user through built-in readback mechanisms or trace recording. Built-in logic such as the JTAG interface, also used for programming, provides read access from all circuit nodes [6]. However, using the JTAG *scan-based* infrastructure to observe the execution in real-time and in situ can be destructive. The execution of the circuit needs to be paused to read out all signals of interest.

The alternative trace *record-replay* approach described in Section 1.3.3 is achieved through the use of Embedded Logic Analyzers (ELA). We describe

ELAs below, followed by similar approaches targeting HLS design flows.

2.2.1 Embedded Logic Analyzers

Embedded Logic Analyzers enable the monitoring of selected hardware signals, in situ and at runtime, through the insertion of trace buffers. The ELAs store the signals of interest cycle-by-cycle and use a trigger unit to choose when to read back these buffers. Traced signals and signals for trigger conditions are chosen before synthesis, while trigger conditions are often allowed to vary after place and route. Some reconfigurability is allowed to avoid recompilation. These signals are presented as waveforms to the designer and labeled using the name of the HDL or the post-fitting (place and route) resource.

Commercial ELA tools such as SignalTap II [2], ChipScope (now called LogiCORE Integrated Logic Analyzer [73]), and Certus [50] use this approach and incorporate multiple optimizations to maximize the use of memory resources. Recent work on ELAs focuses on resource optimization and incremental instrumentation. The latter is concerned with the reduction of compilation time by creating a field-configurable trigger network overlay [24, 43].

2.2.2 HLS Verification & Source-Level Debug

When debugging circuits created using an HLS flow, analyzing signals at the hardware level is challenging. The user provides a C specification and ultimately implements a circuit into FPGA, therefore, debugging should be performed in the context of the original source code. In related work there are

two main approaches: HLS verification and HLS source-level debug infrastructures. The first category is based on On-Chip Monitors (OCMs) which are created to automatically identify inconsistencies between the execution of the generated hardware and the software-like specification. Work in this category includes Assertion Based Verification (ABV) [20, 35, 63], Control Flow Integrity (CFI) Verification [9], and Embedded Signature Monitoring (ESM) [13].

Instrumentation for HLS source-level debug, on the other hand, has the objective of allowing the user to observe a step by step execution of the code, as introduced in Section 1.3.

Although both verification and source-level debug infrastructures have different objectives, both approaches require instrumentation or hardware resources in order to capture, store, or monitor the behaviour of the user's design.

2.3 HLS On-Chip Monitors

On-Chip Monitors (OCMs) are runtime verification tools which can target different behaviours for analysis. OCMs automatically recognize unexpected behaviour in the generated hardware and notify the user of such events. Counter reactions can be implemented in order to correct the errors detected, although work on reactive OCMs, to our knowledge, has not yet been developed for HLS. Relevant work in this area is presented below.

2.3.1 Assertion Based Verification

Although this approach is commonly used when designing hardware at the RTL level, related work for HLS is more limited. Examples of HLS flows with the capability to recognize assertion statements have been developed by upgrading the tool to support assertions for simulation [63], to synthesize them into custom OCMs [35], or by performing source-to-source transformations into synthesizable constructs [20].

Work on temporal assertions by Ribon in 2011 [63] made use of simulations and HDL built-in assertion statements. These were inserted automatically during program synthesis by translating behavioural assertion statements into *temporal assertions*. Temporal assertions are assertions with timing specifications, therefore described in a HDL. This translation included information on operation scheduling, data availability, and uses the corresponding signals associated with the high-level variables.

In order to complement the verification and debugging approach given by assertions in software, HLS can also make use of timing information to generate OCMs. In 2011 Curreri's ABV approach [20] used Impulse-C and the *time* library to automatically insert *clock()* calls in the circuit and evaluate the time passed between calls, which is then compared with a predefined value to determine timing compliance. This is only applicable to the CoDeveloper HLS tool developed by Impulse [44]. The inclusion of the *time* library and other standard libraries is a feature that is missing from most HLS tools.

In 2014, Hammouda proposed two flows that can be applied to any HLS

tool. The first flow is to automatically synthesize assertion statements as OCMs [35], while the second flow automatically generates a CFI verification OCM infrastructure [9]; the latter is explored in the next subsection. The ANSI-C assertion synthesis flow differs from previous work in that this creates an FSM and Datapath, independent from that of the user circuit. This requires access to the internals of the tools' source code, something that is not assumed by Curreri, but necessary in order to target any HLS tool by using the CDFG (Control and Data Flow Graph) representation.

2.3.2 Control Flow Integrity Verification

Control Flow Integrity (CFI) in software is a safety property to detect attacks that provoke unintended software behaviours. In a broader sense, CFI verification for programmable hardware can help detect unintended circuit behaviours when compared to the high-level source code. For this purpose, only control flow information is required.

The CFI application of Hammouda's [9] work uses the same independent circuit approach seen in [35], which is only feasible if there is access to modify the internals of the HLS tool. This circuit takes the *STATUS* of the user circuit as its input and recognizes control flow discrepancies during execution. The *STATUS* signal is the name given by the authors to a generic set of signals that indicate the state of execution of the synthesized circuit; i.e. state of an FSM, and/or flags in a Status Register (SR). This approach then relies on static analysis of the CDFG of the program and the generation of a fairly complex OCM architecture.

At the same time, the OCM architecture uses an I/O Control Unit

(IOCU) to verify that register loads are performed in the corresponding states, providing both control flow and I/O timing behaviour monitoring. The proposed flow targets any HLS tool, although experimental results are done using a combination of GCC and GAUT [19].

2.3.3 Embedded Signature Monitoring

More recent work from Chen [13] makes use of Embedded Signature Monitoring (ESM). In ESM, the synthesized hardware is extended with the instrumentation for signature generation. Signatures are produced from several circuit state signals, including memory interface signals (address, data in, data out), and FSM and datapath registers. This work uses a co-simulation approach, in which the source code is instrumented both for hardware and software execution; the two versions are executed in tandem to generate and verify signatures using the software execution as the golden model. Several challenges were addressed in this work, including memory address matching between SW and HW, LFSR implementation for signature generation, and area optimization through instrumentation resource binding.

2.4 HLS Source-Level Debug

HLS source-level debugging allows the designer to find and analyse unexpected circuit behaviour in-situ and by inspecting the high-level source code. This debugging approach borrows part of the hardware debugging methodology that allows the visualization of the state of the circuit by inserting instrumentation to record a set of signals of interest into a collection of

memories. After running the design, these memories will contain a history of the execution which can then be read and the behaviour of the circuit can be reconstructed. For this to be useful to the designer in an HLS workflow, the signals have to be related back to source-level variables and statements.

The user can then, offline, replay a step-by-step reproduction of the execution of the program to look for unexpected behaviour, iterating through the complete runtime by pausing program execution, collecting data from the trace buffers, and then resuming the execution. The size of the memories, often called *trace buffers*, and their efficiency in the use of storage, determines the observability of the program. Better observability improves the probability of finding the root cause of the unexpected behaviour. In order to allow greater coverage of the execution history, the trace buffer memories have a rollover behaviour, meaning old buffer entries are evicted for each new write access.

2.4.1 JHDL Debug

Previous work presented in 2003 developed the debugging infrastructure for the JHDL-based Sea Cucumber (SC) framework [39]. This project presented the basis for a debug system that relates the hardware signals of synthesized circuits with the high-level language statements. Special attention was put on allowing software optimizations, since most compiler passes can substantially modify the execution schedule and variable nomenclature. Instrumentation for the SC framework made use of device specific readback mechanisms and additional debug circuitry.

2.4.2 LegUp Debug

Recently, two similar approaches were presented for the C-based LegUp HLS framework, namely Inspect[12] and Goeders' HLS Debugger [27]. The latest version of the LegUp repository contains code from both of these projects [59]

Inspect

Inspect allows single-stepping through either HW cycles or source code statements by using SignalTap II for data capture and storage, and a database to relate signals to LLVM's [45] Intermediate Representation (IR) nodes, and to Verilog. Inspect works as both a debugging infrastructure and an OCM due to its capability of comparing the software execution with the RTL simulation, but more importantly by integrating a *discrepancy detection flow*. These features allow for RTL verification at the same time as providing the user with the source-level debugging information. On the other hand, due to Inspect's use of proprietary circuitry for instrumentation (i.e. SignalTap II), the debugging infrastructure is not optimized for HLS. Signals for instrumentation must be selected manually and are limited by the available on-chip RAM.

Goeders' HLS Debugger

Parallel work presented similar features with the insertion of a custom debugging system during the synthesis process [29], instead of using SignalTap II. This work and subsequent research [28, 30] collected in [27], has focused

on optimizing resource utilization, allowing multiple thread instrumentation, and successfully obtaining longer replay window lengths. This translates into more lines of code available in a step-through interface comparable to *gdb*.

The architecture of this instrumentation includes the following modules:

Debug Manager This module is the communication manager, receiving and transmitting commands from and to the workstation in order to start data collection routines, start/pause/stop execution, etc. Communication is done through a RS232 serial connection with custom commands.

Stepping and Breakpoint Unit This unit enables or disables a clock buffer, controlling the execution of the user circuit. The user is allowed to put a breakpoint in the source code in the same way as it's done in software IDEs. This breakpoint then translates into conditions triggered by a certain circuit state, memory address, or chosen variable value.

Memory Arbiter The instrumentation can take control of the main memory controller in the circuit and use it to retrieve all memory values. The arbiter is used to grant access to either the user circuit or the instrumentation.

State Encoder One of the values recorded into the trace buffer is a state identifier. This is used to relate the execution with one or more statements scheduled concurrently. However, the architecture of the synthesized circuit uses a one-hot encoding which needs be reencoded or compressed to be stored in the trace buffer.

Trace Recorder This is the bulk of the instrumentation. Besides containing the memory blocks to store data and states, this includes a Signal-Trace Scheduler. Only signals relevant to the state in execution are stored, and due to previous knowledge of these signals sizes and schedule, it is possible to rearrange them and make better use of the memory space.

This instrumentation and its optimizations rely heavily on LegUp's memory architecture and synthesis flow. Most of the debugging system stands between the main module and the main memory controller. This LegUp architecture is advantageous for the debugging system, but often impacts design flexibility and performance when compared to Vivado's distributed memories, representing a bottleneck for some array accesses.

2.4.3 Event Observability Ports

The most closely related work to the work of this thesis is on Event Observability Ports (EOPs). Work on EOPs has focused on adding points of connection in the circuit [51] to extract data. It further showed a method of providing these access points through *source-to-source transformations* [53], followed by a set of experiments to demonstrate their feasibility by not causing significant impact to circuit performance and resource utilization [52].

In the encompassing thesis [54] the researcher presented additional methods to instrument variable pointers and to add compatibility with both Vivado HLS and LegUp. This work, however, did not consider the C-level instrumentation of the trace buffers or associated circuitry, and is mainly focused on guaranteeing that this observability does not significantly impact

the original behaviour of the circuit.

This work remarks that source-level instrumentation adds reasonable overhead to the user's circuit. In section 5.1 we compare the results from EOP instrumentation to our own approach, which includes trace buffers and the buffer access circuitry. In appendix B we show a method to insert EOPs, different from the one found in [54], and show the results of additional experimentation.

The main drawback of this implementation is that even though this brings the instrumentation to a higher level of abstraction, it is still necessary to perform significant transformations in RTL or rely on proprietary tools. This work suggests the use of custom logic or ELAs in combination with the EOPs in order to add the triggering logic and trace buffers. In [51], a problem of buffer unbalancing is pointed out, this occurs when the EOPs are connected to individual Event Observability Buffers (EOBs). Due to varying refreshing rates for each variable, these buffers are often not used very efficiently. In response, the authors presented the Relative Assertion Rate (RAR) metric and its use for the allocation of a relative size of EOB per variable by using dynamic analysis [51].

2.5 Source-to-Source Transformations

Source-to-source transformation is the manipulation of lines of code, statements, and expressions. In order to allow this, a piece of code or a collection of source code files need to be read into a data structure. The Abstract Syntax Tree (AST) representation is used for this purpose in compilers. An AST

is elaborated following a dictionary and a set of syntactic rules, and creating nodes representing blocks of code, statements, expressions, and symbols. Once parsed, modifications to the AST can be either translated into a lower-level representation, or unparsed in order to generate the original code with the modifications.

2.5.1 ROSE Compiler Infrastructure

The ROSE compiler infrastructure is an open source project by the U.S. Department of Energy with the goal of designing code optimization tools. Through the repository, ROSE provides an Application Program Interface (API) for the inclusion of compiler techniques and optimizations, and also the infrastructure to develop source-to-source transformation tools for custom purposes.

The ROSE API provides a parsing tool that supports multiple files and preprocessing of the source code. A set of query functions allows finding AST nodes according to different attributes and classes; these queries can be very specific or generic in order to do a custom post-filtering of the statements of interest once the query is finished. Statement generation using the ROSE API requires creating the expressions bottom-up, incrementally generating the desired line of code or statement. When unparsing, the ROSE API allows the designer to run a full set of tests called Sanity Check, not only to make sure the AST is unparsed correctly but to make sure it is consistent.

Figure 2.2 presents the Node hierarchy with the main Nodes of interest used in this work. These nodes are represented in an Intermediate Representation (IR) code specific to ROSE called Sage III, and automatically

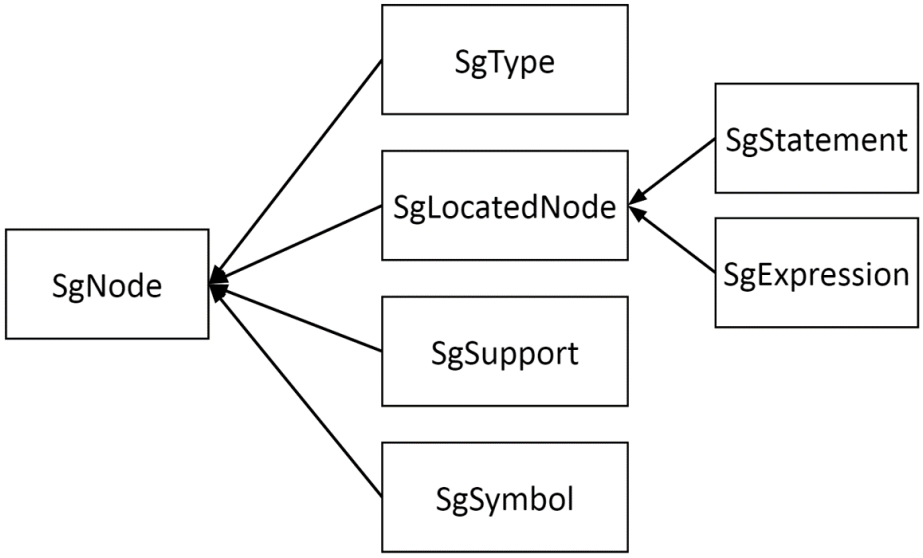


Figure 2.2: Hierarchy ROSE AST Nodes

generated using ROSETTA [62]. The *Sg-* prefix is part of the IR nomenclature.

- An *SgLocatedNode* is a node with a specific location in the source files, also called instructions.
  - *SgExpressions* are instructions that have *return* values, such as operations, and variable/function references.
  - *SgStatements* are any type of instruction, such as control flow modifiers (SgIfStmt, SgForStatement), declarations, and expression statements which are made out of one or multiple *SgExpressions*.
- *SgType* nodes do not have a specific location in the source files, but are used to define each variable type.

- *SgSymbol* nodes are shared and unique for each variable, function, enumerator, label, or any other symbol in the source code.

- *SgSupport* nodes represent all other type of nodes, such as attributes, modifiers (constant, volatile, etc.), and higher level data structures including ROSE projects, files, and tables for internal use by ROSE.

Each node has a set of attributes that indicate its branches or leaves and the characteristics of that class of node. Figure 2.3 is the listing of an example node taken from one of the benchmarks used for the experiments described in this thesis.

Code Transformations

One example of using the ROSE Compiler Infrastructure with HLS applications is a tool flow that can automatically parallelize loops in C/C++ programs that use pointer arithmetic [71]. This work also distributes dynamically allocated data structures between on-chip memory, adding support for dynamic memory allocation calls. Both transformations are performed at the source level, resulting in generic implementations that can be put through various compatible HLS tools.

Code Restructuring

Other work targeting HLS flows has been done in order to optimize the synthesis results through source-level transformations. In [48] and [47] the authors studied the use of loop transformations and pre-optimized templates to aid the user in the process of circuit optimization. Future work, however,

```
pointer:0x1c75f40
SgAssignOp
/home/legup/compileRose/array_elim/chstone/adpcm/adpcm.c 241:7
IsTransformation:0
IsOutputInCodeGeneration:0
Expression type: int

SgNode* p_parent : 0x1c5e890
bool p_isModified : 0
bool p_containsTransformation : 0
$CLASSNAME* p_freepointer : 0xffffffffffffffff
Sg_File_Info* p_startOfConstruct : 0x1c1d198
Sg_File_Info* p_endOfConstruct : 0x1c1d1f0
AttachedPreprocessingInfoType* p_attachedPreprocessingInfoPtr : 0
AstAttributeMechanism* p_attributeMechanism : 0
bool p_containsTransformationToSurroundingWhitespace : 0
bool p_need_paren : 0
bool p_lvalue : 0
bool p_global_qualified_name : 0
Sg_File_Info* p_operatorPosition : 0x1c1cf30
SgExpression* p_lhs_operand_i : 0x1c29cb8
SgExpression* p_rhs_operand_i : 0x1c29d20
SgType* p_expression_type : 0
SgExpression* p_originalExpressionTree : 0
```

Figure 2.3: Example ROSE AST Node

is aimed towards automatic *code restructuring* using source-to-source transformation tools to replace the user's code with a more suitable construct, according to the findings of this work.

Others

There is a great amount of research in source-to-source transformations for software applications, which is often relevant for HLS. This is a great advantage of using HLS with the language subset approach. Relevant work in

this area has been presented using the ROSE compiler infrastructure, with two notable examples.

In [46], the researchers present a code *outliner*. This is a transformation tool capable of generating modular kernels out of whole programs without affecting performance. This is targeted towards parallel computing applications since it includes the use of OpenMP to further optimize the kernels. Moreover, due to recent advances in the use of OpenMP and Pthreads for HLS [16] this work can be applicable to HLS.

Similar work without using ROSE can also be found for parallelization optimizations embedded within HLS tools such as SPARK [34] and ROCCC [33]. Work on source-to-source transformations on specific programs such as X10 [40] have also contributed in developing optimization tools at this level of abstraction. In [40], researchers recently presented methods to incorporate Loop Unrolling, Inlining, Stack allocation, and Loop Invariant Hoisting into the ROSE API.

2.6 Summary

This chapter described the previous work in this area. The objective of this thesis and referenced work is to bridge the gap between software programmers and FPGA implementations. A debug infrastructure is necessary to achieve this goal and it needs to work at the design level (C code), using the information from the implementation level (Hardware signals). Capturing those signals is possible through the use of instrumentation, which can be specified and inserted at various levels of the design. Source-level

instrumentation, however, offers greater portability than other levels of instrumentation, and previous explorations of this approach have been able to achieve this with low impact to the user's circuit performance and area.

Chapter 3

Source-Level Instrumentation

An important part of an effective in-system debug infrastructure is the instrumentation that provides access to internal signals in the design. Such infrastructure needs to consume as little area as possible, and result in the least intrusion possible into the user design. While previous work proved the feasibility of adding EOPs to the user's circuit at the source-level, the work in this thesis includes the trace buffer and associated circuitry. This is beneficial for portability, eliminating the need for RTL editing, and allowing further optimization during synthesis.

This chapter presents our instrumentation that provides both control flow and data capture capabilities, as well as the methods we use to insert this instrumentation. In Section 3.1, the debug framework is presented, which is necessary to insert this instrumentation, and also to be able to retrieve and interpret the data related to the original source code. In Sections 3.2 and 3.3, each proposed feature of the debug framework instrumentation is presented with the corresponding method for insertion (Control Flow and Data Capture) and code examples of the resulting source code.

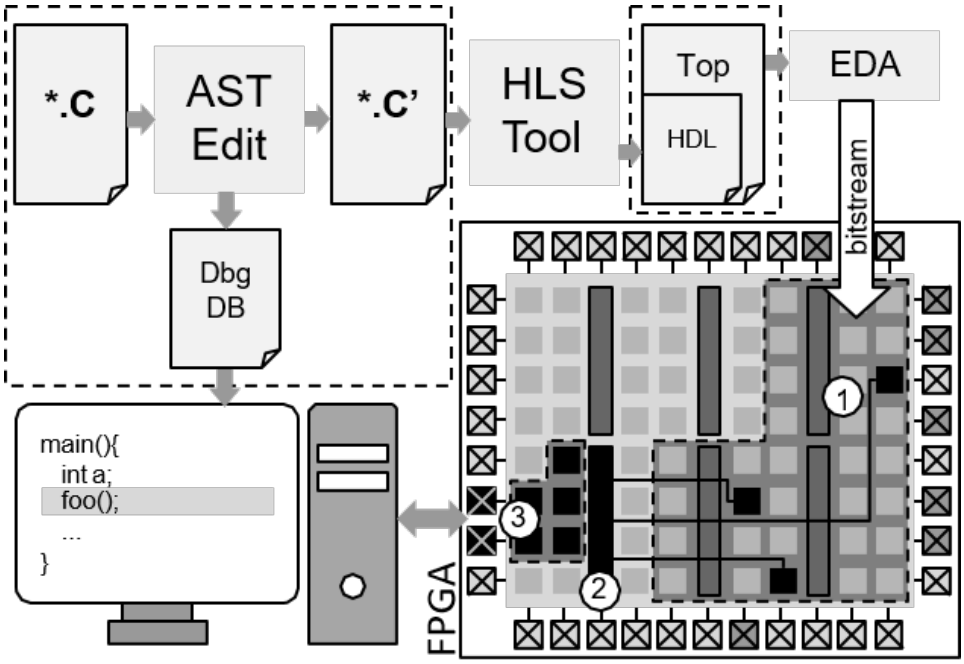


Figure 3.1: HLS In-System Debug Framework with Source-Level Instrumentation

3.1 Source-Level Debug Framework

Figure 3.1 shows our overall in-system debug framework. Starting at the top-left, the original user C code is parsed into the Abstract Syntax Tree (AST) representation. Instrumentation is automatically inserted using the custom tool built using the ROSE source-to-source compiler infrastructure API [61]. Simultaneously, the database is created, mapping the IDs used in these new statements to user's code constructs, statements and/or variable symbols. The last stage in the ROSE compiler unparses the AST to a modified set of C source files.

Initial stages in HLS tools can also perform AST parsing and editing,

meaning these could be modified to incorporate an instrumentation stage, and would, therefore, not require unparsing. Keeping the instrumentation and synthesis separate has an insignificant impact on processing time, while favoring portability.

The modified C code is then compiled using the HLS tool of choice. In our experiments this is either Vivado HLS or LegUp. The synthesized RTL description is then instantiated by a generic top level module that contains elements of the debugging infrastructure (i.e. communication with the workstation, clock buffering, circuit resetting), and is then put through the EDA compilation flow to generate the FPGA bitstream containing the user's circuit and the instrumentation.

3.1.1 Instrumentation

The inserted instrumentation, as seen in Figure 3.1, consists of ① a network that taps off of key signals in the user design, ② a collection of memories, referred to as trace buffers, which are used to store a history of the behaviour of these signals, and ③ the serial connection between the chip and the debug workstation to retrieve the data after the circuit has run. The amount of history available depends on the size of the trace buffers, as well as the efficiency in which data is stored in the trace buffers [30]. This trace record-replay technique is preferable over on-line debugging, where signals are retrieved on every step of execution, because the latter restricts the circuit from running at-speed.

An Integrated Development Environment (IDE) may provide a software-like debugging experience, as described in related work [29], by communi-

cating with the instrumentation provided in this work. The IDE allows the user to run the design at speed, stop at a pre-determined breakpoint, and then retrieve the current value of variables of interest. To provide more information, the trace also contains a *history* of variable values and control-flow information; the larger this history, the easier it will be for the designer to narrow down the root cause of a bug. The code version used through this IDE is the original code. The instrumented code is only for internal use of the HLS process.

In the following subsections we describe how the designs are instrumented, first, to record control-flow information, useful for Control Flow Integrity (CFI) verification. Then, as the main objective is to provide a software-like debugging infrastructure, we provide an extension of the instrumentation to record variable data information and provide a trace readback mechanism.

3.2 Control Flow

We first consider instrumentation that provides sufficient data for an IDE to replay circuit execution, allowing the designer to understand how the design behaved while it was running. Such data can also be used for CFI verification, to find discrepancies between the recorded path and the designer's expectation, by comparing the captured trace to a software simulation, or to a Control Flow Graph (CFG) generated using static analysis.

3.2.1 Control Constructs

In our system, each control construct in the AST of the original code is instrumented. Each of the control constructs is a collection of statements (e.g. For-loop bodies, if-true and if-false bodies, While-loop bodies, Function definitions). At the same time, a database is populated with unique IDs mapped to those constructs. As the circuit executes, each time a control construct is encountered, the instrumentation stores an ID of that construct in the trace buffer.

In the ROSE compiler framework, control constructs can be identified using the SgBasicBlock node class. SgBasicBlocks differ from the Basic Blocks found in compiler frameworks and do not follow the same definition. The Basic Block definition used in compiler frameworks refers to a sequence of lower-level instructions (using the compiler's IR) that only allows one entry and one exit point. An SgBasicBlock, on the other hand, is a sequence of statements of the original source code and does not have this restriction.

3.2.2 Code Example

Listing 3.1 shows an example of this instrumentation. In this example, every execution of the function call *pushDbgCF(<ID>)* "pushes" the ID of the construct of interest into the trace buffer. Lines 1-8, 10, and 13 are the instrumentation code; lines 1-8 are inserted in the global scope while lines 10 and 13 are prepended in each control construct. Line 1 creates the Trace Buffer. A monolithic approach (single trace buffer) was chosen, rather than multiple buffers, to avoid the need for buffer balancing and

Listing 3.1: C Control Flow Instrumentation

```
1  volatile int TRACEBUFFER[TRACESIZE];
2  unsigned int bufIndex=0;
3  void pushDbgCF(int ID){
4      TRACEBUFFER[bufIndex++] = ID;
5      if (bufIndex==TRACESIZE){
6          bufIndex=0;
7      }
8  }
9  void foo(){
10     pushDbgCF(CDBGID_STMNT1);
11     ...
12     if(...){
13         pushDbgCF(CDBGID_STMNT2);
...        ...
```

event-sequence reconstruction. The trace buffer is configured as a circular memory, meaning old buffer entries are evicted. Lines 2, 3-7 provide a trace buffer index and access function; The *pushDbgCF(<ID>)* function contains only the buffer write and a conditional statement to avoid invalid addressing. Experimentally, we have found that this if-statement does not affect the store routine latency when synthesized.

3.3 Data Capture

In order to build a debugging infrastructure, the instrumentation needs to capture the data that is produced while the circuit is running. The second instrumentation strategy we consider provides the ability to gather data regarding the history of variable values over the run of the circuit. Such data can be used in conjunction with an IDE such as that in [27] to provide the ability for the user to observe values in variables using the original code, to help understand the overall behaviour of the circuit.

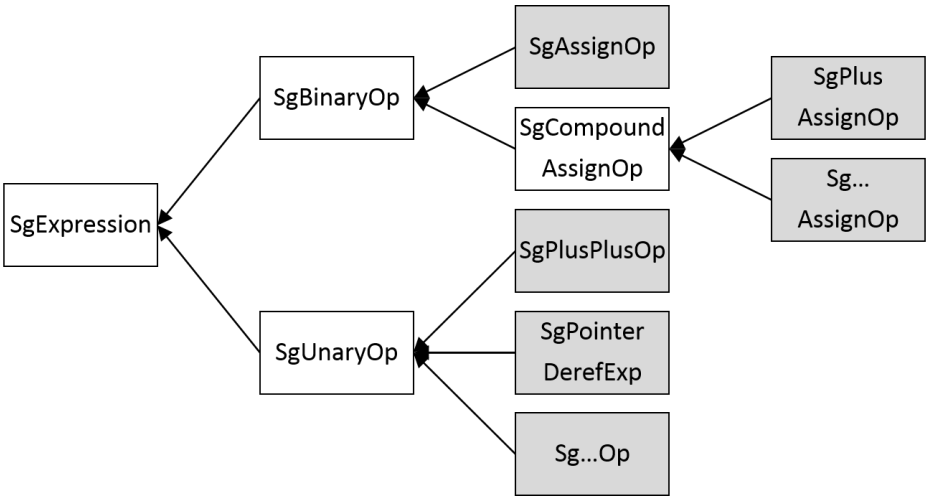


Figure 3.2: Hierarchy of Assignment Statements

3.3.1 Assignment Statements

In the Data Capture strategy, each location in the code in which a variable is updated is assigned a unique ID. An ID is assigned to basic assignments (*a=b*, *a=fn()*), compound assignments (*a+=b*), and unary operations (*a++*). More complex statements are split recursively, and IDs are assigned to those basic statements. Figure 3.2 shows various examples of splitting higher-hierarchy classes (left), until the basic statements of interest are found (shaded blocks on the right). For each assignment statement found in the original code, the transformation tool inserts a call to *pushDbg(<data>),<ID>)*.

Implicitly, this instrumentation acts as *triggering* circuitry. Triggering circuitry specifies when and which signals to store in the trace buffer. In a custom HDL-based approach, a trigger condition needs to be specified for signals of interest, and inserted at design time or incrementally using

spare resources [24, 43]. In HLS instrumentation, triggering conditions are implicitly defined by the assignment statements, meaning that signals are only stored when the state machine reaches the state where its value is modified.

Note that instrumenting for data capture will also provide control flow information. As long as each control construct contains at least one data capture, we will have sufficient data to reconstruct the control flow, meaning this method subsumes that in Subsection 3.2. If a basic block does not contain a data capture (which we did not encounter in our benchmarks) then control flow logging for this particular block can be added. This saves trace buffer space, allowing more updates to be stored in a fixed amount of memory.

3.3.2 Code Example

Listing 3.2 is an example of this technique; lines 1-17, 23, 26 and 28 represent the instrumentation. In this example, two trace buffer access functions are used – one for 32-bit quantities (Line 4) and one for double-length quantities (Line 8). A casting statement using pointer indirection is used in line 26 in order to support storing floating-point data types in the same buffer. Experimentally, we found that having multiple operations in the store routine (i.e. cast, shift, or, and) does not affect the latency of the synthesized hardware.

Listing 3.2: C with Data Instrumentation and Readback

```
1  volatile Ulong TRACE[TRACESIZE];
2  unsigned int bI = 0;
3  volatile Ulong traceOut;
4  void pushDbg(int data,int ID){
5       TRACE[bI++] = ((ULong)ID) << 32 | ((UInt)data);
6       ...//Conditional for invalid addressing
7  }
8  void pushDbgLong(long dataL,int ID){
9    TRACE[bI++] = (((Ulong)ID)<<32) | ((UInt)(dataL>>32));
10   ...//Conditional for invalid addressing
11   TRACE[bI++] = (((Ulong)ID)<<32) | ((UInt)dataL);
12   ...//Conditional for invalid addressing
13 }
14 void traceUnload(){
15   Unload: for (int i = 0; i < TRACESIZE; i++)
16     traceOut = TRACE[i];
17 }
18 int main(){
19     int temp[SIZE];
20     double result;
21     ...
22         temp[i]=fn(); //Assignment Type: int
23         pushDbg(temp[i], CDBGID_STMNT);
24     ...
25         result += temp[i]*3.14159f; //Assignment Type: double
26         pushDbgLong(*(long*)&result, CDBGID_STMNT2);
27     ...
28     traceUnload();
29     return result;
30 }
```

3.4 Trace Readback

The instrumentation techniques described above do not consider any trace reading mechanism. One approach is to instrument the synthesized memory in Verilog by tapping into the Address, Data, and Control signals connected to it. This would affect the high-level compatibility of the rest of the instrumentation, although not drastically since every memory primitive has very similar interfaces. However, to keep the level of abstraction at the source level, it is possible to insert a new function call that will unload the trace memory.

A read-back of the trace buffer must be triggered by an event. This causes the circuit execution to be paused and all data to be sent over the serial connection to the user's workstation. In our implementation, rather than inserting the communication controller in the C code, we halt execution and expose the contents of the trace buffer to be transmitted by a generic serial communication controller in the top module. A simple RS232 interface was implemented for communication between the workstation and the circuit. In HLS, the readback-trigger events coincide with the use of breakpoints; wherever the user sets a breakpoint in an IDE, a trace-unloading routine needs to be inserted.

3.4.1 Code Example

Listing 3.2 shows an example in which the output port declaration and *trace-unloading* function definition are in lines 3 and 14-17, respectively. Line 28 calls the trace-unloading routine. Although such a call can be added

anywhere in the code, care must be taken to avoid adding these calls inside latency sensitive blocks, since interfaces can timeout. Preferably, unload calls can be inserted after a critical section. A call to *traceUnload()* is added by default before the *main* return statement.

3.5 Trace Reconstruction

Trace reconstruction is the step necessary to relate the trace, obtained from the circuit, with the database created during the source transformations. The generated AST needs to be stored and not regenerated, this is due to indeterministic address assignments during source code parsing. Even though the resulting AST structure is deterministic, the addresses associated to each node can vary; in Figure 2.3 these values can be seen as *pointer:0x1c75f40* for that specific node, and that node's parent and branches (i.e. *p_parent*, *p_lhs_operand*, *p_rhs_operand*).

The database contains a one-to-one relation between the *Identifiers* assigned for each *pushDbg(<ID>)* call and the pointer value to one statement. These statements are control assignments or assignment statements and, therefore, contain information about the exact location of the instruction and its attributes.

3.6 Summary

This chapter described our approach for inserting instrumentation into a user design. Unlike previous work, our instrumentation includes the trace buffer and associated circuitry in order to capture the data flow of the execu-

tion. In addition, this Chapter presented the Control Flow instrumentation, allowing the user to opt for a less detailed, less invasive debugging flow alternative.

A set of transformation tools built using the ROSE compiler infrastructure were developed to test these different features through the experiments described in Chapter 5. All the tools follow the same automatic process described in Section 3.1 and can be easily configured to perform either Control Flow or Data Capture instrumentation insertion, change the buffer size, or to enable/disable optimizations like the one described in the following chapter.

Chapter 4

Array Duplicate Minimization

The instrumentation approach described in this work can lead to unnecessary data duplication. The fact that we use instrumentation at the source level, however, provides a unique opportunity to address the duplication. This chapter explains this opportunity for optimization, followed by two strategies to improve data observability and minimize data duplication. Afterwards, an example is used to demonstrate the advantages of the proposed Array Duplicate Minimization (ADM) strategy, and to present the metrics that allow us to quantify how ADM benefits observability.

4.1 Array Duplication

Data duplication occurs every time a value that lives in a user's circuit structure (memory or register) is stored in the trace buffer; some transient values may only exist in wires and won't be duplicated. Until the variable or array entry is changed, the value lives in two places: the user circuit and the trace buffer. This duplication is reasoned by the fact that, ultimately, the trace buffer will contain a history of the values and not just the current content.

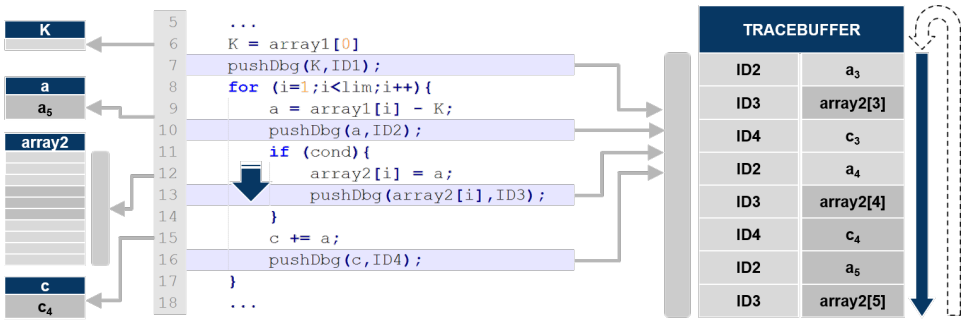


Figure 4.1: Data Duplication

In the case of variable arrays, updates are less frequent than for scalar variables. Therefore, array duplicates are more likely to be evicted from the trace buffer before the next update, meaning, these did not contribute to the trace history.

Figure 4.1 is an example of a section of user code with instrumentation inserted at the source-level. The original code, along with the instrumentation, is shown in the center of the figure. The boxes on the left side of the figure represent on-chip storage that is part of the user circuit; this storage holds values of user variables and arrays as the code executes. The right side of the figure shows the trace buffer; this is part of the debug instrumentation which stores a history of variable and array values. The arrows on the right indicate the rollover behaviour of the memory in which new entries replace the oldest. Blocks with values that are contained in both the user circuit and in the trace buffer are shaded; this represents duplication that we seek to reduce or eliminate in this chapter.

4.2 Merged Instrumentation

In this strategy, we selectively identify arrays in the user circuit. For all updates to these arrays, we do *not* add instrumentation code to store values in the trace buffers. Instead, we modify the *trace_unload* function, so that when the trace buffer is read, these select arrays are unloaded, along with the trace buffers. The IDE software can then use this information to determine the latest values for all elements in the array. Intuitively, this strategy will result in fewer trace buffer updates, meaning a longer history of the other user variables can be stored (recall that the trace buffers are configured as circular buffers, and when full, old data is evicted). In addition, fewer trace buffer updates lead to less contention for memory, reducing the impact on circuit latency.

However, this method may make it impossible to reconstruct control flow history. Recall from Section 3.3 that the data capture instrumentation strategy subsumes the control flow strategy assuming there is at least one data update in each control flow construct. By removing writes to the trace buffer, we increase the likelihood that the IDE does not have sufficient information to reconstruct the control flow. Although we can add control flow instrumentation within the affected control flow constructs, this eliminates the advantage of this strategy.

4.3 Old Value Store

In this strategy, we also modify the *trace_unload* function, so that when the trace buffer is read, select arrays are unloaded. For all updates to the

Listing 4.1: C with Array Duplicate Minimization

```
1  ...//Buffer, index, functions and output declaration
2  void traceUnload();
3  volatile ARRAY1TYPE_T array1[ARRAY1SIZE];
4  int main(){
5      ...
6      pushDbg(array1[x],CDBGID_LINE)
7      array1[x]=a;
8      ...
9  }
10 void traceUnload(){
11   traceUnload: for (int i = 0; i < TRACESIZE; i++)
12     traceOut = TRACEBUFFER[i];
13   array1Unload: for (int i = 0; i < ARRAY1SIZE; i++)
14     traceOut = (long)array1[i];
15 }
```

Listing 4.2: Alternative to Store Old Value

```
...      // This method favors Common Subexpression Elimination
6        int *temp_ptr_array1_803_866 = &array1[x];
7        pushDbg(*temp_ptr_array1_803_866,CDBGID_LINE);
8        *temp_ptr_array1_803_866 = a;
...      ...
```

selected arrays, however, we add instrumentation to store the *old* value of the array element rather than the new value. Although this does not reduce pressure on the trace buffers, it does increase the history available for the elements of the select arrays, while also providing information that can be used to reconstruct control flow. A longer history for these array values means that an engineer using the IDE would have more information to help locate the root cause of observed incorrect behaviour.

4.3.1 Code Example

Example instrumentation for this technique is shown in Listing 4.1; the array *array1* is moved to global scope, and is read by the *traceUnload* function.

The function call in line 6 in Listing 4.1 is inserted before the assignment statement. This has two benefits: it allows us to reconstruct the control flow (as long as there is at least one access in each control construct) and it extends the effective history of the array data. These modifications are uniquely possible at the source-level, where array reference and pointer dereference expressions are explicitly identifiable. An alternative way to call the *pushDbg* routine that we found gives better synthesis results is shown in Listing 4.2.

Note that only the array accesses are affected; writes to scalar variables are performed as previously. This technique does not need to be applied for every array in the user circuit. The selection of arrays for which this technique is applied is currently done manually. The selection of arrays is important, since if we indiscriminately apply the technique, resource utilization may rise. The investigation of automatic array selection policies is left as future work. Note that this technique *could* be applied to individual variables in the user circuit (not just arrays), however, we feel that the overhead in doing so would be too high.

4.3.2 Trace Example

To better understand the idea behind ADM to extend the effective history of array data, consider the example of Table 4.1. This table shows the history of both array and variable updates for a particular run of the MIPS simulator program from the CHStone benchmark set [38]. Each row of the table shows an assignment statement as well as the line number where it can be found in the source code. The table includes an event number (EX)

Table 4.1: MIPS Benchmark Example History

Event	Line	Data Assignment Statement
E0	142	pc = pc + 4;
E1	258	dmem[DADDR (reg[RS] + (ins & 0xffff))] = reg[RT];
E2	141	ins = imem[IADDR (pc)];
E3	142	pc = pc + 4;
E4	241	reg[RT] = reg[RS] + (ins & 0xffff);
E5	141	ins = imem[IADDR (pc)];
E6	142	pc = pc + 4;
E7	258	dmem[DADDR (reg[RS] + (ins & 0xffff))] = reg[RT];

Table 4.2: Trace Buffer Contents

ID	Data
pc_142	pc+4
dmem[x]_258	reg[RT]
ins_141	imem[x]
pc_142	pc+4
reg[y]_241	reg[RS] + ...
ins_141	imem[y]
pc_142	pc+4
dmem[z]_258	reg[RT]

(a) Data Capture

ID	Data
pc_142	pc+4
dmem[x]_258	$\text{dmem[i]}_{\text{old}}$
ins_141	imem[x]
pc_142	pc+4
reg[y]_241	$\text{reg[j]}_{\text{old}}$
ins_141	imem[y]
pc_142	pc+4
dmem[z]_258	$\text{dmem[k]}_{\text{old}}$

(b) Data Capture with ADM

for each statement; these will be used in the following discussion.

Assuming a trace buffer size of eight entries, Table 4.2 shows the contents of the trace buffer after the code is executed for two scenarios: without ADM (4.2a) and with ADM (4.2b). As described previously, each entry in the trace buffer contains an ID and the data update. In the table corresponding to the ADM scenario, three data elements are shaded; these elements are memory entries (rather than scalar variable accesses). In these cases, the *Old Value* of the array element is stored in the trace buffer as described above.

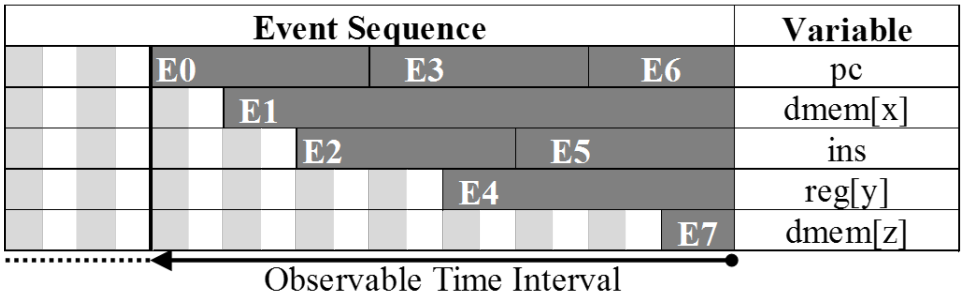
4.3.3 Trace Analysis

After running the circuit, when the execution is replayed using the off-line software tool, the amount of information available to the user is limited by the information that can be obtained through the readback routine.

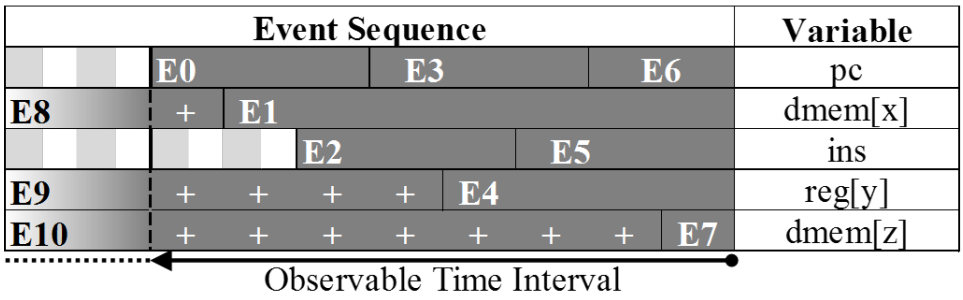
Figure 4.2 shows a representation of the history of data values available *for each variable* accessed in the code. The horizontal axis in each diagram represents time, going backwards from the instant when execution of the code was halted. The left-most record (E0) is the oldest record and the right-most record (E7) occurred immediately before the readback routine.

The top diagram shows the scenario without ADM. In that case, a history of three values is available for PC; these values correspond to the updates labeled E0, E3, and E6 in Table 4.1. Correspondingly, only one value is available for reg[y] from E4, and so on.

The lower figure shows the scenario with ADM. In this case, a history of two values is available for reg[y]: the value corresponding to update E4 (retrieved from the array itself) as well as an older value (update E9 in



(a) Trace without ADM



(b) Trace with ADM

Figure 4.2: Trace Buffer Analysis

the diagram). This older value represents the assignment that was made to reg[y] *before* the update E4. In this way, the ability to retrieve the old values for array accesses increases the amount of history available, providing more information to the user, possibly allowing him or her to have a more complete picture of the operation of the circuit.

4.4 Observability

We expect that an increase in observability, provided by ADM, would have a beneficial impact on the debugging experience. With more observability the user may be able to replicate the bug and find the root cause of that

behaviour using fewer iterations.

Using Figure 4.2 we illustrate how much of the circuit's execution we can observe with that trace sample. In order to measure this, we identified a unit of time. For our instrumentation, even though timestamps or execution cycles are not stored, the events' sequence can be used as a proxy to measure time.

4.4.1 History Coverage

In the sequence of events represented in Figure 4.2a, the entry for event E1 was recorded earlier than the entry for E7. To quantify this relation, we use the number of entries as a proxy to measure the *time* that each entry is valid.

After the entry for event E1 was recorded, six more entries were recorded before readback. The entry for event E1 is valid for seven entries, counting its own.

On the other hand, E7 was the last event recorded in the trace buffer before readback, which means zero entries were recorded after it. The entry for event E7 is valid for one entry.

Consequently, on Figure 4.2a we can observe variable *dmem[x]* for 7 entries, and variable *dmem[z]* for 1 entry. We call this a variable's *History Coverage*.

Multiple Entries per Variable

Some variables have multiple entries, e.g. variable *pc* has three entries (E0, E3, and E6). In those cases, an entry is valid until readback or until the

Table 4.3: MIPS History Metrics Example

Variable	History Coverage	History Coverage w/ ADM	History Coverage Increase
pc	8	8	0
dmem[x]	7	8	1
ins	6	6	0
reg[y]	4	8	4
dmem[z]	1	8	7
Total	**26**	**38**	**12 (46%)**

next update. The entries for events E0 and E3, for example, are valid for three entries, while the entry for E6 is valid for two entries. The *History Coverage* for variable *pc* is the sum of those values. With this in mind, we obtain column 2 of Table 4.3.

Old Value Entries

In the sequence of events represented in Figure 4.2b, the *History Coverage* needs to consider the use of old array values. These are values in entries associated to variable-arrays that were selected during the ADM optimization (i.e. E1, E4, and E7); we call these *Old Value Entries*.

For example, the *Old Value Entry* provided by event E1, is a value that was assigned to *dmem[x]* on a previous event (E8). That entry, however, is only valid for one entry. We cannot assume the observability of *dmem[x]* further back, because the oldest recorded entry is for event E0, meaning any event previous to E0 could have modified *dmem[x]*.

On the other hand, the value read directly from the circuit memory synthesized for *dmem[x]* corresponds to the value assigned to *dmem[x]* on

event E1. This value is, therefore, valid for the same seven entries as the previous case without ADM, from event E1 until readback.

The *History Coverage* for variable *dmem[x]* is now the sum of the valid entries of the *Old Value Entry* and the value read directly from the circuit memory. Column 3 of Table 4.3 shows the *History Coverage* when using ADM with arrays *dmem[]*, and *reg[]* selected for optimization. Column 4 shows the increase in observability for each variable.

4.4.2 Total History Coverage

The total history coverage at one point in the execution is the sum of the *History Coverage* of all variables. The last row in Table 4.3 obtains this value for the sample trace of the MIPS program, without ADM in column 2, with ADM in column 3, and the overall increase in column 4.

Overall, from Figure 4.2 we can measure a 46% *History Coverage* increase, achieved with the use of ADM selecting arrays *dmem[]*, and *reg[]*.

These metrics are collected automatically during the experiments presented in Section 5.4.1.

4.5 Summary

Data duplication is caused by the use of the *record and replay* technique for debugging instrumentation. The trace buffer needs to temporarily store the same values contained in the circuit's structures. This duplication can be reduced by using ADM.

ADM is an optimization focused on removing data duplication for select

arrays. This is done by adding instrumentation to read directly from arrays and recording the old value that was stored in the array index whenever it is updated. This allows us to reconstruct the control and data flow of the execution without duplication and with additional information; arrays selected in ADM are fully visible and old values may provide more observability.

In this chapter we presented two trace examples, with and without the use of ADM. We showed the trace buffer contents from the execution of the MIPS benchmark in the CHStone benchmark suite in those two cases. Finally, this chapter introduced a way to determine how much of the circuit execution can be observed with the content of the trace buffer at one point in the execution. This observability metric is used in the experiments for ADM in the following chapter.

Chapter 5

Experiments and Results

In this chapter, we experimentally evaluate the size and performance overhead caused by the insertion of instrumentation at the C level. Experiment 1 is inspired by [52]. As in the previous work, our goal is to determine the overhead required by our instrumentation techniques. Unlike the previous work, our instrumentation includes the trace buffer memories and associated circuitry, while the instrumentation in [52] is aimed at connecting key points in the circuit to EOPs.

Experiment 2 evaluates the overhead added by the instrumentation when all identified statements of interest are instrumented. Control Flow and Data Capture alternatives are explored for latency and resource overheads.

Experiment 3 investigates partial instrumentation, and measures the variability that is observed. A variability agent is identified and a new configuration of the HLS tool is evaluated using the same experiment. ADM experimentation follows a similar methodology, again focusing on latency and resource overhead.

Table 5.1: Latency Impact Comparison for ADPCM Single Assignment

Tool	Min	Avg	Max	Stdev
EOP[52]-Vivado HLS	-15.00%	0.00%	3.10%	1.00%
Data Capture-Vivado HLS	-15.41%	-0.09%	10.39%	2.40%
Data Capture-LegUp	-0.02%	1.99%	20.00%	4.90%
Control Flow-Vivado HLS	-15.41%	-0.35%	0.69%	2.33%
Control Flow-Legup	1.08%	6.03%	21.25%	5.85%

Table 5.2: LUT Impact Comparison for ADPCM Single Assignment

Tool	Min	Avg	Max	Stdev
EOP[52]-Vivado HLS	-1.10%	0.20%	4.80%	0.60%
Data Capture-Vivado HLS	-3.51%	-0.09%	3.17%	0.69%
Data Capture-LegUp	-4.41%	0.04%	16.38%	1.84%
Control Flow-Vivado HLS	-3.30%	-0.01%	2.47%	0.94%
Control Flow-Legup	-4.27%	0.38%	3.69%	1.36%

Table 5.3: FF Impact Comparison for ADPCM Single Assignment

Tool	Min	Avg	Max	Stdev
EOP[52]-Vivado HLS	-7.40%	0.20%	10.70%	1.20%
Data Capture-Vivado HLS	-6.62%	0.22%	8.26%	1.50%
Data Capture-LegUp	-4.45%	0.75%	19.79%	2.20%
Control Flow-Vivado HLS	-6.25%	0.40%	3.55%	1.27%
Control Flow-Legup	-4.73%	0.72%	4.88%	1.79%

Table 5.4: LE/LC* Impact Comparison for ADPCM Single Assignment

Tool	Min	Avg	Max	Stdev
EOP[52]-Vivado HLS†	N/A	N/A	N/A	N/A
Data Capture-Vivado HLS	-1.33%	1.18%	6.89%	0.98%
Data Capture-LegUp	-4.43%	0.54%	18.91%	2.04%
Control Flow-Vivado HLS	-1.58%	1.73%	3.81%	1.22%
Control Flow-Legup	-4.59%	0.62%	4.33%	1.57%

* Logic Elements (LEs) for LegUp and Logic Cells (LCs) for Vivado HLS

† EOP results do not include this metric

5.1 Experiment 1: Single Instrumentation Point

We first consider the impact of instrumenting single points of interest. In this experiment, we cycled through all possible *data* and *control* instrumentation points (assignments and control constructs) and instrumented each in isolation. We assumed a 256-entry trace buffer and performed our experiments using the CHStone benchmark suite. As in [52], we were unable to compile three of the CHStone benchmarks for Vivado HLS without hand-coded modifications, and so they were omitted from the experiments.

Tables 5.1-5.3 show the results for one benchmark circuit, ADPCM. Experiments using Vivado HLS used the Xilinx Artix-7 XC7A35T [72] FPGA device while LegUp uses the Altera Stratix V 5SGXEA7 [5]. The changes in latency, number of LookUp-Tables (LUTs), number of Flip-Flops (FFs), and number of Logic Elements (LEs) or Logic Cells (LCs) are shown. LEs and LCs are, respectively, the names given by Altera and Xilinx to a unit of configurable fabric that includes LUTs and FFs. For each, we tabulate the minimum, maximum, average, and standard deviation over all instrumentation points in the circuit. Results are shown using both LegUp and Vivado HLS, using both Control Flow and Data Capture, as well as results from [52]. The latency results represent a slowdown if positive or speedup if negative.

Comparing the first two rows of numbers in Tables 5.1-5.3 (EOP and Data Capture-Vivado HLS), we see that the overhead results of our approach match those in [52] closely. The results for the other circuits in the benchmark suite also matched the trends presented in [52]. This suggests

that the extra overhead due to the trace buffers and extra control logic for single assignments do not significantly affect the size and performance of the instrumented circuit. Comparing these to the third rows of Tables 5.1-5.3 (Data Capture-LegUp), we see the latency overhead is higher if LegUp rather than Vivado HLS is used. We expect that this is because the version of LegUp we used groups all global variables in a single monolithic memory, meaning it is more likely that trace buffer writes interfere with variable updates in the user circuit. On the other hand, Vivado HLS instantiates multiple memory arrays and controllers, meaning it can better tolerate the extra memory updates added with our instrumentation.

Rows four and five of Tables 5.1-5.4 have the results of these experiments for control flow instrumentation. The area and performance overhead results for single control flow statements are similar to those for data capture. Table 5.4 shows the changes in LEs or LCs according to the HLS tool; LegUp or Vivado HLS respectively. These numbers represent circuit size overhead in one single value.

5.2 Experiment 2: Complete Instrumentation

For our second experiment, we considered the overhead if *all* control constructs or assignment statements are instrumented simultaneously. The latter would be required in a flow similar to the one found in [27] which provides access to all variables within a window of interest. Such a strategy is important if we wish to provide a debug experience similar to software; in software debuggers such as GDB, access to all variables is provided. How-

ever, due to the overhead of hardware debug instrumentation, the common HLS debug workflow will focus on a selection of variables. CFI verification scenarios would also benefit from complete control flow information. In this experiment, we gather separate results for control flow instrumentation (as described in Section 3.2) as well as data capture instrumentation (as described in Section 3.3) using a 256-entry buffer.

Numbers for latency are presented first. Latency is a critical metric for debug instrumentation. A modification that affects latency could also remove the root cause of a system-level bug, or create one, making the instrumentation detrimental for the debugging process. A change in latency can be caused by the addition or removal of states in the state machine or stages in the datapaths, or different assignments of resources during allocation and binding. These translate to changes to the critical path, the creation of helpful/harmful delay slots, or affect (cause or prevent) conflicting use of resources. This type of bug, which may appear or disappear due to the insertion of instrumentation, is classified as a Heisenbug [32]. For source-level instrumentation, maintaining a low impact on latency is a greater challenge because there is less control over resource allocation and operation scheduling; these tasks are relegated to the HLS tool. During our experiments, all benchmarks executed correctly before and after instrumentation, meaning we did not observe Heisenbugs.

5.2.1 Latency Impact

The latency results for control and data instrumentation are shown in Tables 5.5 and 5.6. Columns 3 and 4 show both the total latency of the resulting

instrumented circuit, and the overhead percentage over the original latency. The percentage is used in order to analyze how the original latency affects the results. Here, instrumentation on circuits with short original latencies tend to generate a higher percentage overhead (i.e. DFADD, DFMUL). However, this is also the case for other benchmarks (i.e. MIPS) as will be described below.

When instrumentation is added, latency tends to increase, partly because source-level instrumentation may interfere with optimizations that optimize away certain operations or signals. When optimized, however, signals are prone to be removed and their behavior is not available during an RTL-instrumented debug. A source-instrumented version will preserve all signals directly relatable to source variables, producing a functionally-identical circuit but possibly interfering with an optimization that could have produced a more efficient design. The impact of this, however, was found to be small.

Control Flow

As in Experiment 1, we see that the latency overhead is more significant when compiling the design with LegUp, due to the increased congestion for access to the single memory caused by the instrumentation. The ADPCM circuit on LegUp has the most significant impact which seems to be caused by a more frequent use of global arrays when compared to other circuits. This increases congestion with accesses to the trace buffer.

Table 5.5: LegUp Cycles (Slowdown)

Benchmark	Original	Control	Data
ADPCM	13221	31585(138.9%)	65308(394.0%)
AES	9193	10892(18.5%)	11538(25.5%)
BF	163925	186862(14.0%)	188983(15.3%)
DFADD	673	3166(370.4%)	1308(94.4%)
DFDIV	1916	2806(46.5%)	2674(39.6%)
DFMUL	224	1113(396.9%)	983(338.8%)
DFSIN	59061	88162(49.3%)	51090(-13.5%)
GSM	4771	6059(27.0%)	5710(19.7%)
MIPS	5035	10892(116.3%)	7805(55.0%)
Average	**28669**	**37949(32.4%)**	**37267(30.0%)**

Table 5.6: Vivado HLS Cycles (Slowdown)

Benchmark	Original	Control	Data
ADPCM	28880	31633(9.5%)	41249(42.8%)
AES	3159	3223(2.0%)	3623(14.7%)
BF	107429	107950(0.5%)	112889(5.1%)
DFADD	405	519(28.1%)	433(6.9%)
DFDIV	1980	2043(3.2%)	1979(-0.1%)
DFMUL	215	284(32.1%)	227(5.6%)
DFSIN	51635	55547(7.6%)	50810(-1.6%)
GSM	3728	3963(6.3%)	3721(-0.2%)
MIPS	2541	3251(27.9%)	5535(117.8%)
Average	**22219**	**23157(4.2%)**	**24496(5.8%)**

Data Capture

Data capture results reveal the same trend of the previous experiments. The impact on latency for LegUp instrumentation is higher.

In more detail, the higher latency overhead found for the MIPS benchmark generated by both HLS tools is caused by multiple concurrent assignments that could not be scheduled efficiently. These assignments belong to decoded signals from the *ins* (instruction) variable, which is split into multiple variables for decoding. A constraint of the trace buffer limits the number of inputs to two on each cycle (inferred dual-port RAM), however, the scheduler also uses available *slots* in the following cycles. This situation is witnessed in other benchmarks but, in this particular case, all available *slots* are occupied. This affects all loop iterations of the program, causing a significant impact. The ADPCM benchmark using LegUp presents an enormous overhead, which is caused by a combination of the previous factors; multiple assignments are scheduled concurrently and the memory bottleneck is significantly stressed. In this case, ADPCM's higher impact over that on MIPS appears to be due to the former's heavier use of global memory as seen with the Control Flow instrumentation.

On the other hand, some benchmarks show unchanged latency or even a speedup of up to 20%. Speedups, although beneficial during normal synthesis, are unsought during debug instrumentation. These imply a change that could cause Heisenbugs. These results, however, suggest there is another consideration that causes scheduling changes; this is further explored in Section 5.3.

Table 5.7: LegUp Complete Instrumentation Logic Elements (Overhead)

Benchmark	Original	Control	Data
ADPCM	22216	26511(19.3%)	29962(34.9%)
AES	26546	27241(2.6%)	30818(16.1%)
BF	14778	16568(12.1%)	17213(16.5%)
DFADD	12324	14555(18.1%)	14161(14.9%)
DFDIV	18809	20209(7.4%)	18771(-0.2%)
DFMUL	8901	10502(18.0%)	8929(0.3%)
DFSIN	39096	44851(14.7%)	45639(16.7%)
GSM	18705	21612(15.5%)	19662(5.1%)
MIPS	6541	9189(40.5%)	8456(29.3%)
Average	**18657**	**21249(13.9%)**	**21512(15.3%)**

5.2.2 Resource Utilization

Area results are presented in terms of Logic Elements in Table 5.7 and in terms of Logic Cells in Table 5.8. Columns 3 and 4 show both the total resources required by the resulting instrumented circuit, and the overhead percentage over the original implementation. In these results, we see the percentages are rather stable. As the number of signals and, therefore, size of the original circuit increase, the instrumentation also increases in order to create the access network from the signals to the trace buffer.

Overall, area overhead when using LegUp is lower than that when using Vivado HLS for data capture. However, the total resource utilization for LegUp is significantly higher even though all optimizations (-O3) were enabled. Outlying results, such as ADPCM, are found to be caused mostly by inlining changes; this will be further explored in Section 5.3.

For implementation, a Stratix V 5SGXEA7 [5] was chosen for LegUp and an Artix-7 XC7A35T [72] FPGA for Vivado HLS. The resource uti-

Table 5.8: Vivado HLS Complete Instrumentation Logic Cells (Overhead)

Benchmark	Original	Control	Data
ADPCM	6764	7064(4.4%)	11305(67.1%)
AES	2386	3936(65.0%)	4562(91.2%)
BF	1645	1642(-0.2%)	4185(154.4%)
DFADD	2826	2716(-3.9%)	3892(37.7%)
DFDIV	2894	3011(4.0%)	3707(28.1%)
DFMUL	1397	1594(14.1%)	2310(65.4%)
DFSIN	11549	12035(4.2%)	15351(32.9%)
GSM	3871	4193(8.3%)	5518(42.5%)
MIPS	1329	1333(0.3%)	2036(53.2%)
Average	**3851**	**4169(8.8%)**	**5874(52.5%)**

lization numbers are presented in Tables 5.7 and 5.8. An increase can be seen between control and data instrumentation. Since data instrumentation requires connecting all active bits of each variable, the multiplexing logic required to share access to the trace buffer is significantly higher than that for control flow. Control flow instrumentation only needs to multiplex *Identifiers* which are constants and can be optimized during logic synthesis.

Comparison with Previous Work

In Tables 5.9 and 5.10 we compare the numbers gathered from our experiments with those from the EOP [52] and Goeders' RTL instrumentation [30], respectively. Goeders' results were obtained using the same Stratix V 5SGXEA7 FPGA as used in our LegUp experiments, while the EOP experiments were performed on a Xilinx Zynq XC7Z020 FPGA [76]; both the Artix-7, used in our experiments, and Zynq devices use 6 input LUTs. Although common debug instrumentation would only target a set of signals, these results with *all* statements instrumented reveal characteristics of the

Table 5.9: Vivado HLS EOP vs Data Capture Overhead

Benchmark	EOP[52]*		Data Capture	
	LUTs	FFs	LUTs	FFs
ADPCM	25%	2%	76%	37%
AES	22%	2%	119%	41%
BF	50%	25%	413%	76%
DFADD	70%	10%	43%	17%
DFDIV	27%	2%	42%	10%
SHA	27%	5%	55%	19%
Average	**37%**	**8%**	**58%**	**25%**

*EOP results do not include a trace buffer or a buffer access network

different approaches.

EOP EOP results taken from [52] are only given in LUTs and FFs overhead percentage for a smaller subset of the CHStone benchmark suite. The numbers for Data Capture in Table 5.9 show the impact caused by the multiplexing logic needed to share access to the trace buffer. The trace buffer itself uses on-chip RAM. The numbers from [52] only represent the impact to the original circuit and do not include the multiplexing logic of the buffer access network. The LUTs overhead in our experiments increases significantly as the number of variables increases, due to the need of more multiplexing logic. The increase in FF numbers may be due to an increase in the number of states and impact on optimizations. The EOP experiments in [52] or [54] do not present latency impact for *all* statements instrumented.

Goeders' The results of using Goeders' HLS Debugger are compared with our results using LegUp in Table 5.10. Multiple factors pose a challenge to make this a direct comparison. In terms of the modules included, Goeders'

Table 5.10: LegUp HLS Debugger vs Data Capture Logic Elements (Overhead)

Benchmark	Goeders'[30]		This thesis*	
	Original	Debug	Original	Data Capture
ADPCM	17900	22244 (24%)	22216	29962 (35%)
AES	16966	18372 (8%)	26546	30818 (16%)
BF	10448	13692 (31%)	14778	17213 (16%)
DFADD	7261	10473 (44%)	12324	14161 (15%)
DFDIV	12233	16018 (31%)	18809	18771 (0%)
DFMUL	4171	6414 (54%)	8901	8929 (0%)
DFSIN	24982	31928 (28%)	39096	45639 (17%)
GSM	9096	11415 (25%)	18705	19662 (5%)
JPEG	35691	38305 (7%)	48354	69110 (43%)
MIPS	3162	4380 (39%)	6541	8456 (29%)
Average	**14191**	**17324 (22%)**	**18657**	**21512 (15%)**

*These results do not include the RS232 communication manager.

results are affected by the inclusion of the RS232 communication manager and the scheduling logic in the trace recorder. Our results do not include either as they focus on the impact caused to the original circuit. Adding the estimated LEs necessary to include the communication manager, which is similar between both approaches, increases the average impact to approximately 21%. However, Goeders' trace scheduler, which allows longer trace recordings in the RTL approach [27], represents a large part of the overhead and has no analogy in our approach.

The comparison between these two approaches is made harder by the signal selection stage of each process. On the source-level approach, variables selected in the original source-code are instrumented regardless of the resulting circuit. An RTL approach, however, instruments an optimized description of the logic, starting with high-level optimization passes, and followed by

logic synthesis optimizations. High-level optimization passes like dead code elimination (DCE) apply to both cases, where "unreachable" instructions are eliminated, including the instrumentation in the source-level approach. However, optimizations that apply constant folding techniques, can result in (a) constant inputs to the trace buffer in a source-level approach, and (b) signals that are not available in hardware and ideally inferred during synthesis.

Therefore, a direct comparison would require listing these cases and applying source-level instrumentation only to the variables for which RTL instrumentation was inserted. This experiment is out of the scope of this work but is proposed as future work, along with the integration of source-level optimizations as discussed in 6.1.

Overall, RTL instrumentation for LegUp often provides observability with lower impact to the original circuit than source-level instrumentation. Additional resources can then be used to optimize this architecture to allow longer trace records. This, however, is the result of tool-specific design and optimizations. Goeders' instrumentation relies heavily on the memory architecture used in LegUp.

5.3 Experiment 3: Partial Instrumentation

The data capture overhead numbers in Tables 5.5 and 5.6 are large; this implies that it may be appropriate to consider instrumenting only a subset of the available signals. Intuitively, and as shown by [52], cumulatively increasing the number of statements that are instrumented increases the cost

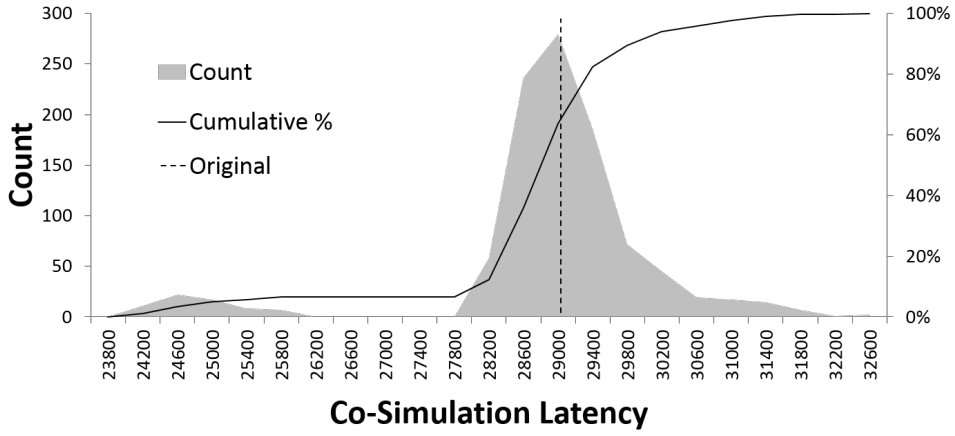


Figure 5.1: Latency Histogram for Variation of ADPCM

of that instrumentation. However, the total cost of instrumenting multiple assignments may be less than multiplying the number of assignments by the cost of instrumenting each single assignment individually, due to resource usage overlap.

For further experimentation, we created a set of 1000 different instances of the ADPCM benchmark, each instance with 10 random statements instrumented, to observe the variability of the results depending on what assignments are chosen. The sample set is a small subset of all possible combinations $\binom{119}{10}$, however, the distribution trend was validated with different set sizes and random selection seeds. Each instance was compiled using Vivado HLS, and hardware simulation latency results for all 1000 instances are shown in the histogram of Figure 5.1. The left vertical axis shows the count of instances that performed with a latency matching the ranges in the horizontal axis. The right vertical axis is the cumulative percentage. From these results, 60.4% of the instances require less than or an

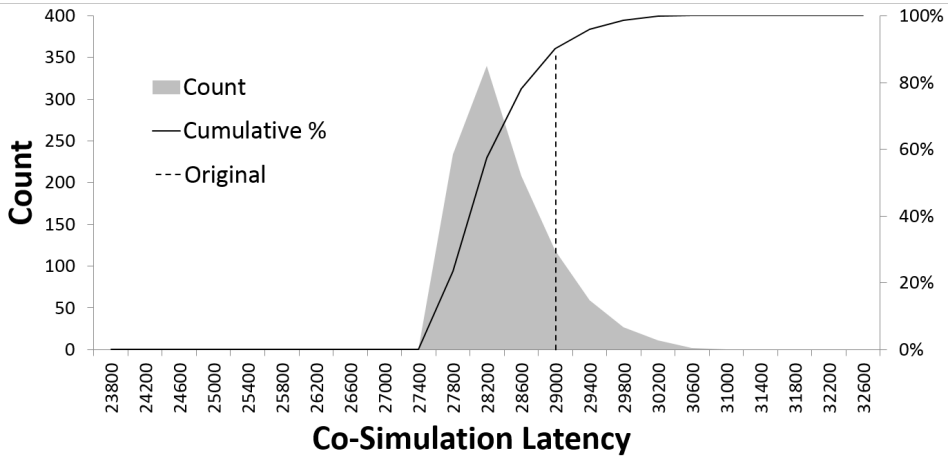


Figure 5.2: Latency Histogram for Variation of ADPCM All-Inlined

equal amount of cycles as the original design (28880 cycles). Average latency is -0.6% lower than the original, and standard deviation is 4.4%. Average resource utilization is 1.7% higher, with a standard deviation of 152 LCs (2.2%).

The observed latency changes were caused, in part, by the considerations mentioned above. However, changes during the inlining optimization of the HLS tool were found to be more significant. The HLS tools can be configured to use a different *inline threshold* or to inline every function in the program. An experiment using the same set of 1000 instances of the ADPCM benchmark with all functions inlined, produced a distribution of latency results with less variation. For the data in Figure 5.1, the standard deviation was 1263 cycles (4.4%), and average 28694 cycles; the always-inlined version of these instances showed a standard deviation of 604 cycles (2.1%), with an average of 28180 cycles (-2.4% lower). This lower latency and less variant latency, however, requires higher resource utilization [41].

Table 5.11: Duplication Metrics

Benchmark	Average Memory Entries	Average Duplicates	Coverage Increase
ADPCM	75.09(29.3%)	41.73(16.3%)	43.76%
AES	112.98(44.1%)	48.36(18.9%)	90.37%
BF	122.61(47.9%)	55.41(21.6%)	91.31%
GSM	76.03(29.7%)	69.23(27.0%)	59.52%
JPEG	76.59(29.9%)	76.14(29.7%)	95.44%
MIPS	23.65(9.2%)	16.13(6.3%)	65.41%
SHA	80.36(31.4%)	80.36(31.4%)	124.47%
Average	**81.04(31.7%)**	**55.34(21.6%)**	**84.28%**

Average resource utilization is 4.6% higher, with a standard deviation of 328 LCs (5.1%).

5.4 ADM Evaluation

In this section we evaluate the ADM strategies. To understand their potential, we first measured how often data is duplicated between memory arrays and the trace buffer. Results for the following experiments on each benchmark program are presented in Table 5.11. We made these measurements using a subset of the CHStone benchmark, excluding the arithmetic domain programs (DFADD,DFDIV,DFMUL,DFSIN) because of their algorithmic/structural simplicity, i.e. read from ROM, compute, and compare; which does not require arrays for computation.

On average, from the second column of Table 5.11, we found that 31.7% of the trace buffer entries are from user memories. These would allow the Merged Instrumentation technique from Section 4.2 to use up to 31.7% more entries for scalar values, depending on which arrays are selected. Moreover,

we found that 21.6% of the trace buffer entries are from user memory entries that are not evicted – multiple assignments to the same entry in an array are counted as one. Therefore, a Data Capture instrumentation with Old Value Store ADM has the potential to reduce the amount of data duplication by 21.6%. Appendix C shows the array access profiles of the CHStone benchmark programs obtained during the implementation of the previous experiment.

5.4.1 Observability

We performed an experiment with the observability metrics proposed in Section 4.4 to evaluate the average increase in history coverage we can obtain from the Old Value Store ADM technique.

We gathered the results in the third column of Table 5.11 based on our experiments on the CHStone benchmarks. This shows that an average of 84.2% more history coverage can be stored in the trace buffer using this technique. This demonstrates the benefit of implementing Old Value Store ADM, suggesting that HLS-generated circuits contain enough identifiable arrays, such that, storing the value being evicted from those arrays can result in a significant contribution to circuit execution observability.

5.4.2 Resource Utilization

To analyze the impact on area overhead of ADM, we measured the resource utilization after place and route of (a) the original ADPCM benchmark, (b) the circuit instrumented with data capture instrumentation, (c) the same instrumentation with our data duplication minimization approach applied

Table 5.12: ADM Resource Utilization for ADPCM Instances

Instance	BRAM	FFs (Overhead)	LUTs (Overhead)
(a) Original	7	5135	9200
(b) Data Capture	9	7448 (45.0%)	14203 (54.4%)
(c) ADM array100	9	7464 (45.4%)	14284 (55.3%)
(d) ADM array24	9	7460 (45.3%)	14288 (55.3%)
(e) ADM (both)	9	7476 (45.6%)	14307 (55.5%)
(f) ADM (13 arrays)	9	7579 (47.6%)	14616 (58.9%)

to one array of size 100, (d) to one different array of size 24, (e) to both arrays, and (f) to all data accesses in the circuit (13 arrays were identified). Table 5.12 shows that it is increasingly less expensive to add multiple arrays, and that memory utilization is kept constant, as expected. Moreover, for the full instrumentation of the ADPCM circuit (f), adding 2.6% more FFs and 4.5% more LUTs to (b) allowed 16.3% duplication to be removed, with 43.76% more history coverage.

5.5 Summary

This chapter presented the methodology and results of various experiments to evaluate the impact caused by the insertion of debug instrumentation at the source level. In our first set of experiments we instrumented single points of interest. This showed that, on average, instrumenting these single points causes low impact on the area and speed of the original designs. This observation, also observed in previous work on EOPs, is extended here to include the insertion of a trace buffer and related circuitry. The second experiment, instrumenting all points of interest, showed how the impact on the original

circuit increases, suggesting that source-level debugging using source-level instrumentation may benefit from prior selection of regions of interest in the source code to avoid impractical overheads. The third experiment, identified the causes of performance overhead when partial instrumentation is applied. Occasionally, states are added to the circuit FSMs to schedule trace buffer writes. However, results would vary significantly due to changes in the resulting code after inlining optimizations.

We also presented experiments to evaluate our ADM approach. The use of ADM in our instrumentation allows us to extend circuit observability with low overhead. Using our metric for *History Coverage* we found that ADM can extend circuit observability by an average of 84%. This is the result of being able to observe the values of select arrays before the time they were evicted.

Chapter 6

Conclusion

High-level synthesis tools promise increased productivity for designers, allowing them to create compute accelerators more rapidly, and test them in the application environment from an early stage of development. This promise, however, will only be realized if the compilers are accompanied by an entire ecosystem including a debugging framework that allows designers to debug their designs in the context of the original C code, while running in silicon.

In-system source-level debugging tools for HLS rely on instrumentation to record the behaviour of the design as it executes, for later interrogation by an off-line software debugger. We showed a source-level instrumentation technique that includes the trace buffer and related circuitry. This instrumentation is inserted by automatic source-to-source transformation tools to record both the control flow and data assignments into trace buffers. The impact on circuit size varies from 15.3% to 52.5% and the impact on circuit speed ranges from 5.8% to 30% when *all* assignment statements are instrumented. The impact on circuit size for single point instrumentation ranges from -4.6% to 18.9% and the impact on circuit speed ranges from -15.4% to 21.3%. A variability agent is identified in the inlining optimization and

evaluated.

During these transformation the tools also create a database that maps source code statements to unique identifiers. Matching the trace captured after execution with the offline database allows mapping the behaviour of the circuit back to the original C code.

We also showed how our design can be optimized to make better use of resources by eliminating up to 31.7% of the data duplication between circuit memories and trace buffers. This helps extending visibility into the execution history, resulting in 84.2% more coverage, which we anticipate will translate into a faster and easier debugging experience, requiring fewer executions to find the root cause of observed incorrect behaviour.

6.1 Limitations

When instrumenting high-level code there are changes that can affect lower level optimizations. The nature of the statements being inserted can especially affect loop optimizations. A noticeable effect is found on loop-invariant code; this consists of variables or expressions inside a loop body that are unaffected by all loop iterations and can, therefore, be moved outside of the loop body. This is called loop-invariant code motion, hoisting or scalar promotion. Adding a function call to *pushDbg()* inside the loop with that expression as an argument creates a false dependency that invalidates the optimization.

In the scope of this work, the impact of source code manipulation in loop-invariant code motion were not considered.

However, there is a considerable amount of work on source-level optimizations, including the use of the ROSE compiler framework for those specific optimizations [40], as well as others, mentioned in Section 2.5. Therefore, these optimizations can be incorporated as a step previous to the instrumentation insertion, either with automatic code transformations or using programmer feedback to indicate the loop-invariant expression.

An addition that can be suggested for a better debugging experience is the inclusion of execution cycles numbers with every entry in the trace buffer. Although this is not necessary for a behavioral representation of the code, this is useful for performance analysis and more advanced debugging. RTL instrumentation techniques [12, 27] implement custom counters in logic for this purpose. However, when using source-level instrumentation, this would have to rely on tool-specific features to incorporate custom Verilog; this approach has been used for timing analysis assertions [20]. A more suitable alternative is the inclusion of the *<time.h>* library as an API for HLS tools. For the HLS tool user, this would mean being able to call functions like *clock()* and *time()* and meaningful macros such as *CLOCKS_PER_SEC* in their programs. For debugging purposes, using these same functions would allow adding timestamps to trace buffer entries.

6.2 Future Work

As mentioned above, the incorporation of timestamps into the trace buffer would be beneficial to expand the capabilities of this approach towards timing analysis and performance debugging. Working with open source HLS

tools to add API support of the *time.h* library could be appealing for developers and suggest a standard for proprietary HLS tools to follow. The work in [20] lays an approach with the basic requirements.

Future work specific to ADM involves developing heuristics to select memories for minimization, as well as quantifying and comparing the performance (e.g. lines of code in a replay window) of this, other types of optimizations, and other types of instrumentation.

Work on EOPs [54] made a compatibility study of other HLS tools, without the need for empirical testing. This was accomplished by identifying the requirements to allow such transformations, and found that only 1 (Shang [80]) out of 12 tools was incompatible; LegUp was also deemed incompatible for not being able to schedule concurrent I/O, but this limitation, although unnecessary for the enhanced instrumentation approach, can be overcome through the modifications explained in Appendix A.

With the upcoming release of Altera's A++ HLS framework [1], empirical testing of RTL generation of source-level instrumented circuits on this tool will be necessary.

Bibliography

[1] Altera. Altera Announces New Spectra-Q Engine for Industry-leading Quartus II Software to Accelerate FPGA and SoC Design. http://newsroom.altera.com/press-releases/nr-altera-spectraq-quartusii-software-fpga-soc.htm, May 2015.

[2] Altera. *Quartus Prime Pro Edition Handbook*, volume 3, chapter 9: Design Debugging Using the SignalTap II Logic Analyzer. November 2015.

[3] Altera. Altera SDK for Opencl. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html, 2016.

[4] Altera. Stratix 10 FPGA and SoC - Overview. https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html, 2016.

[5] Altera. Stratix V FPGAs - Overview. https://www.altera.com/products/fpga/stratix-series/stratix-v/overview.html, 2016.

[6] H. Angepat, G. Eads, C. Craik, and D. Chiou. Nifd: Non-intrusive fpga

debugger – debugging fpga 'threads' for rapid hw/sw systems prototyping. In *2010 International Conference on Field Programmable Logic and Applications*, pages 356–359, Aug 2010.

[7] ARM. big.LITTLE Technology: The Future of Mobile. `https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf`, 2016.

[8] David Bacon, Rodric Rabbah, and Sunil Shukla. FPGA Programming for the Masses. *Queue*, 11(2):40:40–40:52, February 2013.

[9] Mohamed Ben Hammouda, Philippe Coussy, and Loic 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. ACM, 2014.

[10] John Bodily, Brent Nelson, Zhaoyi Wei, Dah-Jye Lee, and Jeff Chase. A Comparison Study on Implementing Optical Flow and Digital Communications on FPGAs and GPUs. *ACM Trans. Reconfigurable Technol. Syst.*, 3(2):6:1–6:22, May 2010.

[11] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 109–116, May 2014.

[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*, Sept 2014.

[13] Keith A. Campbell, David Lin, Subhasish Mitra, and Deming 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.

[14] A. Canis, S.D. Brown, and J.H. Anderson. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014.

[15] Andrew Canis, Jongsok Choi, et al. LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, September 2013.

[16] Jongsok Choi, S. Brown, and J. Anderson. From software threads to parallel hardware in high-level synthesis for FPGAs. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 270–277, Dec 2013.

[17] J. Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on SDC formulation. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 433–438, 2006.

[18] Intel Corporation. Intel Completes Acquisition of Altera. `http://www.intc.com/releasedetail.cfm?ReleaseID=948014`, December 2015.

[19] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. *GAUT: A High-Level Synthesis Tool for DSP Applications*, pages 147–169. Springer Netherlands, Dordrecht, 2008.

[20] John Curreri, Greg Stitt, and Alan D. George. High-level Synthesis of In-circuit Assertions for Verification, Debugging, and Timing Analysis. *Int. J. Reconfig. Comput.*, 2011:1:1–1:17, January 2011.

[21] Luka Daoud, Dawid Zydek, and Henry Selvaraj. *Advances in Systems Science: Proceedings of the International Conference on Systems Science 2013 (ICSS 2013)*, chapter A Survey of High Level Synthesis Languages, Tools, and Compilers for Reconfigurable High Performance Computing, pages 483–492. Springer International Publishing, 2014.

[22] Eclipse CDT. Eclipse CDT (C/C++ Development Tooling). `http://www.eclipse.org/cdt/`, 2016.

[23] Chris Edwards. Growing pains for deep learning. *Commun. ACM*, 58(7):14–16, June 2015.

[24] F. Eslami and S. J. E. Wilton. An adaptive virtual overlay for fast trigger insertion for FPGA debug. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 32–39, Dec 2015.

[25] Zhe Fan, Feng Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster

for high performance computing. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 47–47, Nov 2004.

[26] Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. An Overview of Reconfigurable Hardware in Embedded Systems. *EURASIP J. Embedded Syst.*, 2006(1), January 2006.

[27] J. Goeders and S. 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*, PP(99):1–1, 2016.

[28] J. Goeders and S. J. E. Wilton. Using round-robin tracepoints to debug multithreaded hls circuits on fpgas. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 40–47, Dec 2015.

[29] J. Goeders and S.J.E. Wilton. Effective FPGA debug for high-level synthesis generated circuits. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014.

[30] 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.

[31] Paul Graham, Brent Nelson, and Brad Hutchings. Instrumenting Bitstreams for Debugging FPGA Circuits. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '01, pages 41–50, 2001.

[32] Michael Grottke and Kishor S Trivedi. A classification of software faults. *Journal of Reliability Engineering Association of Japan*, 27(7):425–438, January 2005.

[33] Zhi Guo, Betul Buyukkurt, Walid Najjar, and Kees Vissers. Optimized Generation of Data-Path from C Codes for FPGAs. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 112–117, March 2005.

[34] Sumit Gupta, Manev Luthra, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Hardware and Interface Synthesis of FPGA Blocks Using Parallelizing Code Transformations. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems*, November 2003.

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

[36] Frank Hannig, Dirk Koch, and Daniel Ziener, editors. *Proceedings of the First International Workshop on FPGAs for Software Programmers (FSP 2014)*, August 2014.

[37] Frank Hannig, Dirk Koch, and Daniel Ziener, editors. *Proceedings of the Second International Workshop on FPGAs for Software Programmers (FSP 2015)*, August 2015.

[38] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[39] 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.

[40] Michihiro Horie, Mikio Takeuchi, Kiyokuni Kawachiya, and David Grove. Optimization of x10 Programs with ROSE Compiler Infrastructure. In *Proceedings of the ACM SIGPLAN Workshop on X10*, X10 2015, pages 19–24, 2015.

[41] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Nazanin Calagar, Stephen Brown, and Jason Anderson. The Effect of Compiler Optimizations on High-Level Synthesis-Generated Hardware. *ACM Trans. Reconfigurable Technol. Syst.*, 8(3):14:1–14:26, May 2015.

[42] E. Hung and S. J. E. Wilton. Speculative Debug Insertion for FPGAs. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 524–531, Sept 2011.

[43] Eddie Hung and Steven J. E. Wilton. Accelerating FPGA Debug: Increasing Visibility Using a Runtime Reconfigurable Observation and Triggering Network. *ACM Trans. Des. Autom. Electron. Syst.*, 19(2):14:1–14:23, March 2014.

[44] Impulse Accelerated Technologies. CoDeveloper from Impulse Accelerated Technologies. `http://www.impulseaccelerated.com/ReleaseFiles/Help/iAppMan.pdf`, 2015.

[45] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–86, 2004.

[46] Chunhua Liao, Daniel J. Quinlan, Richard Vuduc, and Thomas Panas. Effective source-to-source outlining to support whole program empirical optimization. In Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li, editors, *Languages and Compilers for Parallel Computing: 22nd International Workshop, LCPC 2009, Newark, DE, USA, October 8-10, 2009, Revised Selected Papers*, pages 308–322. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[47] J. Matai, D. Lee, A. Althoff, and R. Kastner. Composable, parameterizable templates for high-level synthesis. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 744–749, March 2016.

[48] J. Matai, D. Richmond, D. Lee, and R. Kastner. Enabling FPGAs for the Masses. In *Proceedings of the First International Workshop on FPGAs for Software Programmers*, pages 15–20, Aug 2014.

[49] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk

Stroobandt. An overview of today's high-level synthesis tools. *Des. Autom. Embedded Syst.*, 16(3):31–51, September 2012.

[50] Mentor Graphics. Certus Silicon Debug. `https://www.mentor.com/products/fv/certus-silicon-debug`, 2016.

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

[52] J. S. Monson and B. Hutchings. Using Source-to-Source Compilation to Instrument Circuits for Debug with High-Level Synthesis. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 48–55, Dec 2015.

[53] J. S. Monson and Brad 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, 2015.

[54] Joshua Scott Monson. *Using Source-to-Source Transformations to Add Debug Observability to HLS-Synthesized Circuits.* PhD thesis, Brigham Young University, March 2016.

[55] Roger Moussalli, Ildar Absalyamov, Marcos R. Vieira, Walid Najjar, and Vassilis J. Tsotras. High performance FPGA and GPU complex pattern matching over spatio-temporal streams. *GeoInformatica*, 19(2):405–434, 2015.

[56] 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*, PP(99), 2016.

[57] R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, June 2004.

[58] NVIDIA. NVIDIA® Tegra® X1 - NVIDIA's New Mobile Superchip. `http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf`, January 2015.

[59] University of Toronto. High-level synthesis with legup. `http://legup.eecg.toronto.edu/`, 2016.

[60] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24, June 2014.

[61] Dan Quinlan and Chunhua Liao. The ROSE Source-to-Source Compiler Infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011.

[62] Dan Quinlan and Bobby Philip. ROSETTA: The Compile-Time Recog-

nition Of Object-Oriented Library Abstractions And Their Use Within Applications . In *Proceedings of the PDPTA'2001 Conference*, 2001.

[63] A. Ribon, B. Le Gal, C. Jgo, and D. Dallet. Assertion Support in High-Level Synthesis Design Flow. In *Specification and Design Languages (FDL), 2011 Forum on*, pages 1–8, Sept 2011.

[64] Mike Santarini. Xilinx ships industry's first 20-nm all programmable devices. *Xcell Journal*, 86:9–11, 2014. First Quarter 2014.

[65] Semico. How an FPGA Approach to Complex System Design Can Improve Profitability: Real Case Studies. Case Studies CC303-12, Semico Research Corporation, Apr 2012.

[66] Scott Sirowy and Alessandro Forin. Wheres the Beef? Why FPGAs Are So Fast. Technical Report MSR-TR-2008-130, Microsoft Research, September 2008.

[67] Calypto Design Systems. Catapult c synthesis. `http://calypto.agranderdesign.com/catapult_c_synthesis.php`, 2016.

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

[69] K. Wakabayashi and T. Okamoto. C-based soc design flow and eda tools: An asic and system vendor perspective. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 19(12):1507–1522, November 2006.

[70] Robert A. Walker and Raul Camposano, editors. *A Survey of High-Level Synthesis Systems*. Springer Science Business Media, 1991.

[71] Felix Winterstein, Samuel Bayliss, and George A. Constantinides. Separation Logic-Assisted Code Transformations for Efficient High-Level Synthesis. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, FCCM '14, pages 1–8, 2014.

[72] Xilinx. 7 series fpgas overview. `http://www.xilinx.com/products/silicon-devices/fpga/artix-7.html`, May 2015.

[73] Xilinx. Integrated Logic Analyzer v6.1: LogiCORE IP Product Guide. `http://www.xilinx.com/support/documentation/ip_documentation/ila/v6_1/pg172-ila.pdf`, April 2016.

[74] Xilinx. SDAccel Development Environment. `http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`, 2016.

[75] Xilinx. Vivado Design Suite User Guide: High-Level Synthesis. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug902-vivado-high-level-synthesis.pdf`, June 2016.

[76] Xilinx. Zynq-7000 All programmable SoC. `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`, 2016.

[77] YXI products. eXCite: C to RTL Behavioral Synthesis. `http://www.yxi.com/products.php`, 2016.

[78] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, 2015.

[79] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. *AutoPilot: A Platform-Based ESL Synthesis System*, pages 99–112. Springer Netherlands, Dordrecht, 2008.

[80] Hongbin Zheng, Swathi T. Gurumani, Kyle Rupnow, and Deming Chen. Fast and Effective Placement and Routing Directed High-level Synthesis for FPGAs. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 1–10, New York, NY, USA, 2014. ACM.

Appendix A

LegUp Interface Directive

Vivado HLS provides a comprehensive set of interfacing options for IP integration in the Vivado logic synthesis flow. LegUp, on the other hand, uses a relatively limited memory-mapped approach, although it is possible to use other LegUp features to generate interface descriptions similar to what is provided in Vivado HLS (e.g. Custom Verilog code insertion [54]). The work in this thesis included a modification of LegUp's source code to allow the creation of I/O ports with validity signals, in order to have a more direct comparison between LegUp and Vivado HLS implementations.

To resemble the format used by Vivado HLS, the implemented LegUp feature supports adding a TCL directive to convert a global variable into an I/O port. Listing A.1 is an example of C source code with EOP instrumentation for the "sum". Listings A.2 and A.3 shows the TCL directives for each tool to convert the "eop_14_sum_16" variable into an output port. Vivado's implementation allows modifying the mode and location. These examples use the default settings to create a port with a validity signal located in the main "function/module".

In LegUp, the value assignments to the specified global variables are scheduled using the System of Difference Constraints (SDC) heuristic [17]

without any resource constraints. Multiple concurrent port writes are allowed by both Vivado HLS and this LegUp implementation. The variables are then synthesized into I/O ports (input-only if no values are assigned to it; output-only if no assignments are made from it).

Listing A.1: C with EOP Instrumentation

```
1 //Non-volatile for output-only in Vivado HLS
2 volatile int eop_14_sum_16;
3 int main (){
4 ...
5     sum += A1[i][k] * B1[k][j];
6     eop_14_sum_16 = sum;
7 ...
8 }
```

Listing A.2: Vivado HLS "directives.tcl"

```
1 #set_directive_interface [OPTIONS=-mode(ap_vld)] <location=main> <port>
2 set_directive_interface eop_14_sum_16
```

Listing A.3: LegUp "config.tcl"

```
1 #set_interface_port <global_variable1> <global_variable2> ...
2 set_interface_port eop_14_sum_16
```

Appendix B

EOP Experiment

A custom source-to-source transformation tool for EOP insertion was developed using the ROSE compiler infrastructure for evaluation of this approach. For Vivado HLS, this was achieved using global variables and the *"interface"* directive to instruct the HLS tool to convert these into port interfaces. Vivado HLS automatically includes a "valid" signal that is true during the state with the variable assignment. On the other hand, a new I/O feature had to be implemented in LegUp in order to allow the same type of transformations to be synthesized, this is explained in Appendix A.

In spite of the exhaustive experimentation presented in [52], our approach was used for an unexplored behavior. An experiment was set to understand the motivation for implementing independent EOBs and, generally, the amount of concurrent port writes that are scheduled by either LegUp or Vivado. The CHStone benchmark [38] was instrumented with EOPs for *all* identified assignments, synthesized and wrapped by a top module that included a set of *Concurrent Port Writes Counters*. These counters keep track of the number of valid EOPs at each execution cycle. Figures B.1 and B.2 show the results of this experiment for all the programs in the CHStone benchmark. The X axis shows the number of EOPs, the left Y axis shows

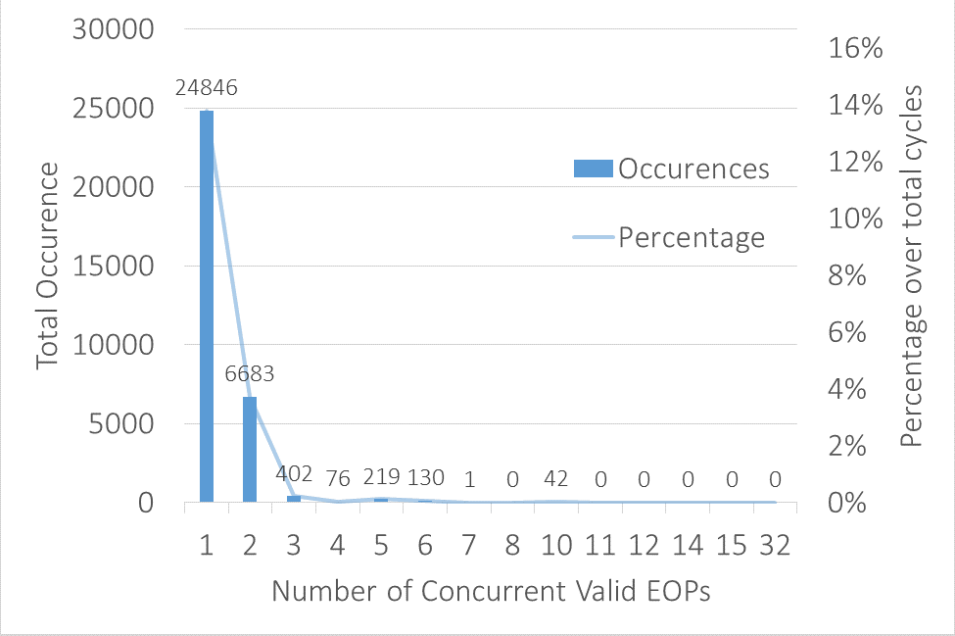


Figure B.1: LegUp EOP Writes

the frequency that the given number of EOPs were found valid, and the right axis is the percentage of that amount over the total number of execution cycles. In general, there is at least 1 valid EOP for 20-30% of the time of the execution. Also, even though HLS is set to identify all possible parallelism and schedule simultaneous assignments, the number of valid ports is generally 1 or 2, and reduces abruptly after 2. Using this information for EOB allocation would require dynamic analysis to guarantee an optimal EOB balancing. Thorough dynamic analysis, however, is not feasible for programs with input-dependent behaviour.

This metric can also be useful for performance profiling, determining the amount of *extracted* parallelism by finding the number of simultaneous *variable* assignments that were scheduled.

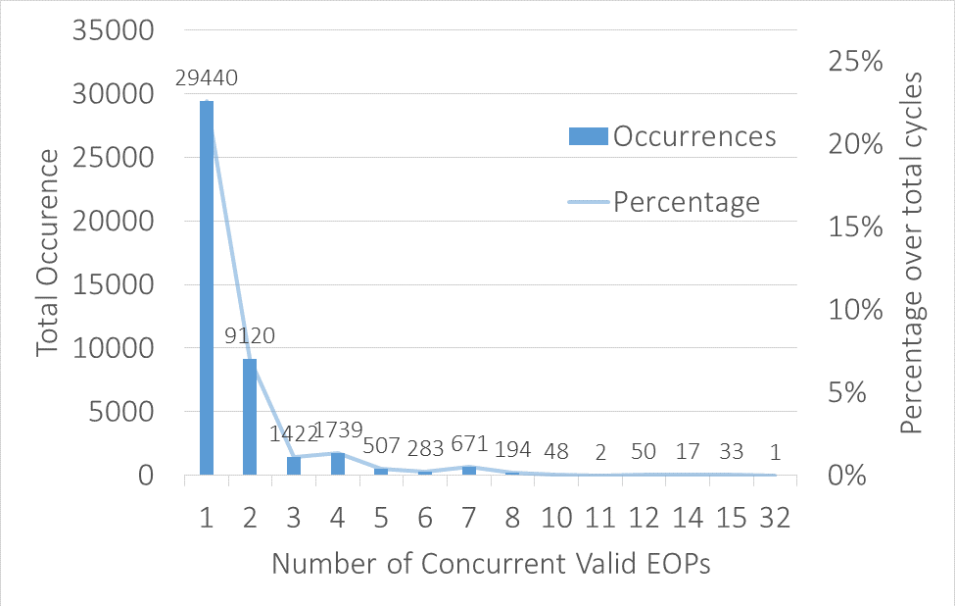


Figure B.2: Vivado HLS EOP Writes

Appendix C

Memory Access Profiles

The setup of the experiments used in chapter 5 that allowed us to measure array duplication in the CHStone benchmark suite can also be used to obtain a Memory Access Profile of the program. For these experiments, each program was first instrumented for Data Capture in C using our ROSE-based transformation tool. Using Vivado HLS, the C code was synthesized and then added to a Vivado project in order to perform logic synthesis. These files can then be used for simulation and obtain behavior details through Verilog *System Tasks and Functions* (i.e. readmem() and writemem()). Using file I/O functions we can *periodically* dump the contents of the trace memory at any time without the need for trace read back routines; this approach is used in other debugging techniques and this data is what is matched with the database. In our experiments, the in-system debug is the main contribution and simulation is only used for benchmark characterization and analysis.

The set of files obtained from each circuit can then be analyzed by reading and matching their contents with the database generated during instrumentation. Trace entries belonging to arrays can be identified as well as the array index used in that statement in order to count the number of dupli-

cates found in memory. Accesses to the same array index are counted only once, since only the last entry is duplicated, or available in both the trace buffer and the circuit memory.

The following graphs indicate the number of unique array entries in the trace buffer on the Y axes and the execution cycles on the X axes. Although some profiles of the CHStone benchmark are generally balanced (e.g. MIPS, ADPCM, SHA), most can be used to determine possible bottlenecks in circuit implementations. In general, it can be seen that memory accesses are common and, as seen in Table 5.11, can represent an average of more than 30% of the assignments carried out in the execution. This is a great motivation for ADM and should be considered as the target for optimizations in future work.

Figure C.1: ADPCM

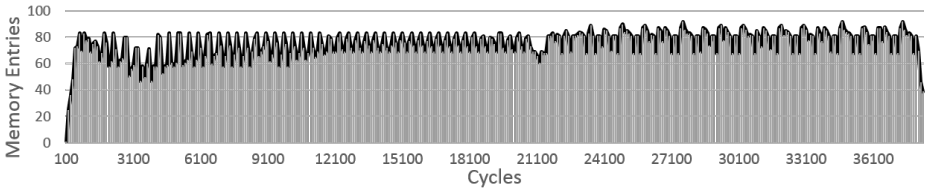


Figure C.2: AES

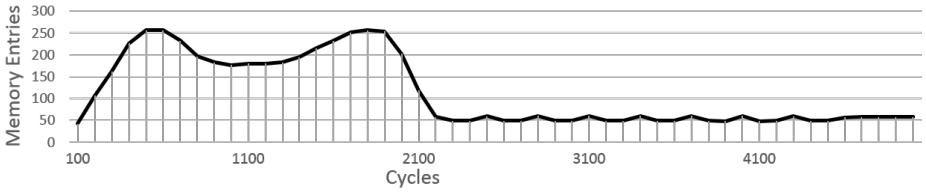


Figure C.3: BLOWFISH

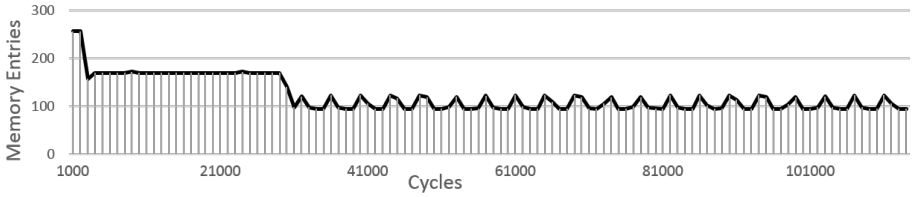


Figure C.4: GSM

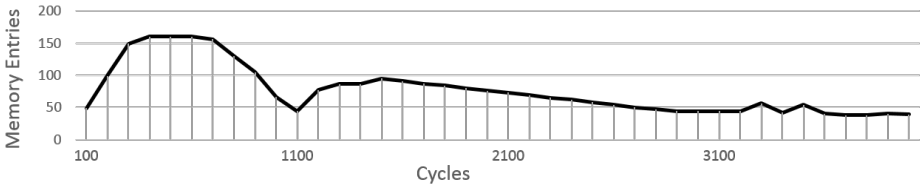


Figure C.5: JPEG

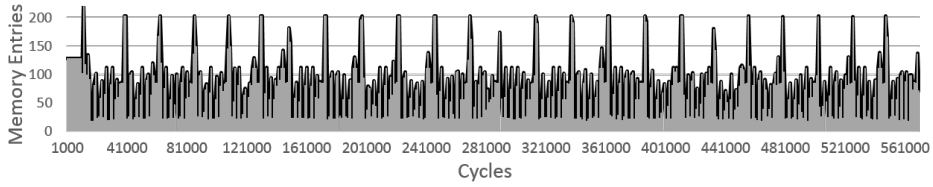


Figure C.6: MIPS

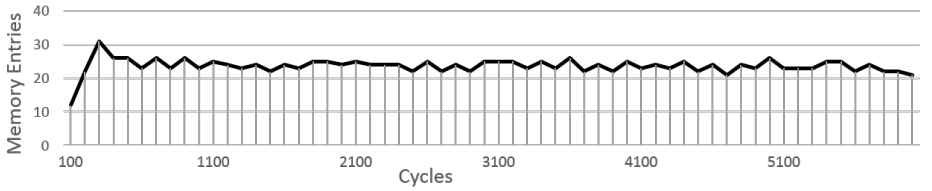


Figure C.7: SHA