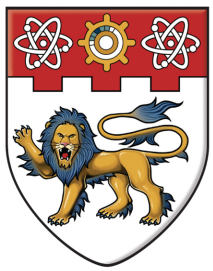

NANYANG
TECHNOLOGICAL
UNIVERSITY

IMPROVING THE PRODUCTIVITY OF HIGH-LEVEL SYNTHESIS BY ADVANCING REUSABILITY AND VERIFIABILITY

LIWEI YANG

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy

January 2017

Acknowledgements

First of all, I would like to thank my supervisors. I would like to thank Dr. Kyle Rupnow for all the hands-on guidance and inspiration. I would like to thank Prof. Douglas Maskell and Prof. Suhaib A Fahmy for the kind support and advice.

I would like to thank all the great people I work with in Advanced Digital Sciences Center, Singapore. I would like to thank Prof. Deming Chen for taking me into this world-class research team to pursue valuable research goals. I would like to thank all the teammates, Swathi Gurumani, Hongbin Zheng, Nguyen Quoc Duy Tan and Augustine Koh for their kind help and the fruitful collaboration. Meanwhile, I would like to thank all my past and present friends and colleagues in the Hardware and Embedded Systems Lab in Nanyang Technological University for making our days in Singapore much more interesting.

Finally, with all of my heart, I wish to express my eternal gratitude to my family for their everlasting encouragement, consolation, understanding and selfless devotion. I want to thank my parents, Wang Mingying (王明瑛) and Yang Shunhe (杨顺和), for your endless love throughout my whole life and the huge support that backs me up when I cannot physically take care of Wanxin and Shaxi. You are the source of everything I become. I would like to thank my wife, Wang Wanxin (王万新), for sharing your whole life and faith with me, and bringing our baby to this world with all you have. It is my greatest blessing to have you as my beloved. I would like to thank my daughter, Yang Shaxi (杨沙溪), for coming into my life as the greatest gift ever and opening a new era of my life.

This achievement is yours.

Contents

List of Figures

List of Tables

Abstract

As the complexity of applications continues to grow to meet user demands, the complexity of hardware platforms continues to grow correspondingly. Thus, the hardware design flow is a critical methodology to handle continued growth in design complexity. Whether targeting application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs), hardware design starts with models and algorithmic specifications in high level languages (HLLs). Following algorithmic specifications, designers perform architectural specification including hardware/-software (HW/SW) partitioning. Next, the hardware design is specified in register-transfer level (RTL) description using hardware description languages (HDL) such as Verilog and VHDL. Computer-aided design (CAD) tools then verify functionality and synthesize the RTL into a netlist of circuits, which is further placed and routed into physical design for the implementation.

As the size (transistor counts) of integrated circuits (ICs) continuously increase, the productivity of the hardware design flow is a major challenge. The productivity of designing in manual RTL has been growing more slowly than the size and complexity of the designs, emphasizing the need for enhancements in the design flow.

High level synthesis (HLS) is an attractive strategy for accelerating the design entry phase by automating transformation of high-level, untimed- or partially-timed specifications to low-level cycle-accurate RTL specifications. HLS improves productivity via both acceleration of design entry through concise HLL descriptions, and software-based behavioral testing. However, despite the significant productivity improvements through HLS, overall design flow productivity still has major challenges. Research in improving design flow productivity include performance monitoring/prediction to identify performance bottlenecks, HW/SW co-design, integration of reusable components, and verification.

In an HLS-based hardware design flow, two particular portions have become particular bottlenecks: integration with reusable intellectual property (IP) components, and verification and debugging. In modern designs, IPs constitute about 70% of

the overall SoC design which contributes to both productivity gain and quality of results (QoR). Manual RTL design requires significant effort to integrate IPs, perform architectural exploration, and implement communication protocols. Although HLS accelerates design entry for important components, the task of integrating the HLS-generated hardware with other IPs remains significant.

Verification and debugging are also critical to HLS-based design flows. Verification consists both of verifying source input and HLS tool correctness. Even in RTL design flows, functional verification consumes over 50% of overall design flow time, but HLS-based design verification is especially challenging. HLS-produced RTL is not human-readable, and thus debugging is particularly time-consuming and cumbersome, back-tracing through hundreds of signals and simulation cycles. Furthermore, verification of the HLS tools is critical; designers must be confident that they can apply any combination of optimizations arbitrarily without concern for whether the tool will produce a functionally correct output. However, HLS tools are different from typical large-scale software; we must verify that the HLS tool output is always functionally correct RTL, not just that the HLS tool runs to completion without error. Therefore, the verification process of the functional correctness of HLS tools has also become a bottleneck that defers the enhancement of QoR, thus preventing HLS from achieving further improved productivity.

In this thesis, we present solutions to address the productivity in HLS-based design flows by effectively integrating reusable IPs, and facilitating the verification and debug process both for HLS-produced designs and the HLS tools. Particularly, we present behavioral-level IP integration in HLS that integrates various IP cores with support for internal and external instantiation, variable- and fixed-latency IPs, and both shared and parallel instantiations. In addition, we present an automated framework to facilitate source-level debugging and verification for HLS-produced designs with cross-layer verification and automated discrepancy detection and analysis. Furthermore, we demonstrate how the debugging and verification framework can also be used to debug and verify the HLS tool itself. Together, this thesis demonstrates critical techniques for addressing productivity in HLS-based design flows to improve the ability to use HLS for large and complex designs.

Chapter 1

Introduction

Integrated circuits (ICs) have had a radical impact on the human world. From personal computers to ubiquitous smart mobile devices and embedded systems, the evolution of hardware creates a variety of powerful devices for innovative and complex applications that revolutionize human life. At the core of this advancement is the semiconductor industry which has been greatly improving the capacity and performance of hardware for decades. However, although chip fabrication continues to deliver faster and more powerful system-on-chip (SoC) platforms, and correspondingly more complex applications, the productivity of effectively designing hardware platforms is an ever-increasing challenge. The productivity of hardware design is falling behind the needs for more and more complex designs. To address these challenges, there has been significant research effort in the hardware design flow to address productivity of hardware design (time-to-market) while retaining required quality of results.

1.1 Hardware Design Flow

The concept of productivity of hardware design is about the efficiency of hardware production, which is closely related to the hardware design flow. With satisfactory quality of results (QoR), given the same amount of effort in the whole design flow, the more the production output is, the higher the productivity is. Hardware

design flows may produce either application-specific integrated circuits (ASICs) or field-programmable gate array (FPGA) configurations. The hardware design flow is a sequence of steps at multiple different design abstraction levels, and design productivity lies in the efficiency of each step of the design flow as well as the interaction of the steps. Designers adopt different methodologies for design specification and implementation to improve productivity, leading to many variations of the general design flow. In order to understand how various design methodologies affect overall productivity, we first introduce the generic design flow, and then we will discuss several of the most common strategies for improving productivity.

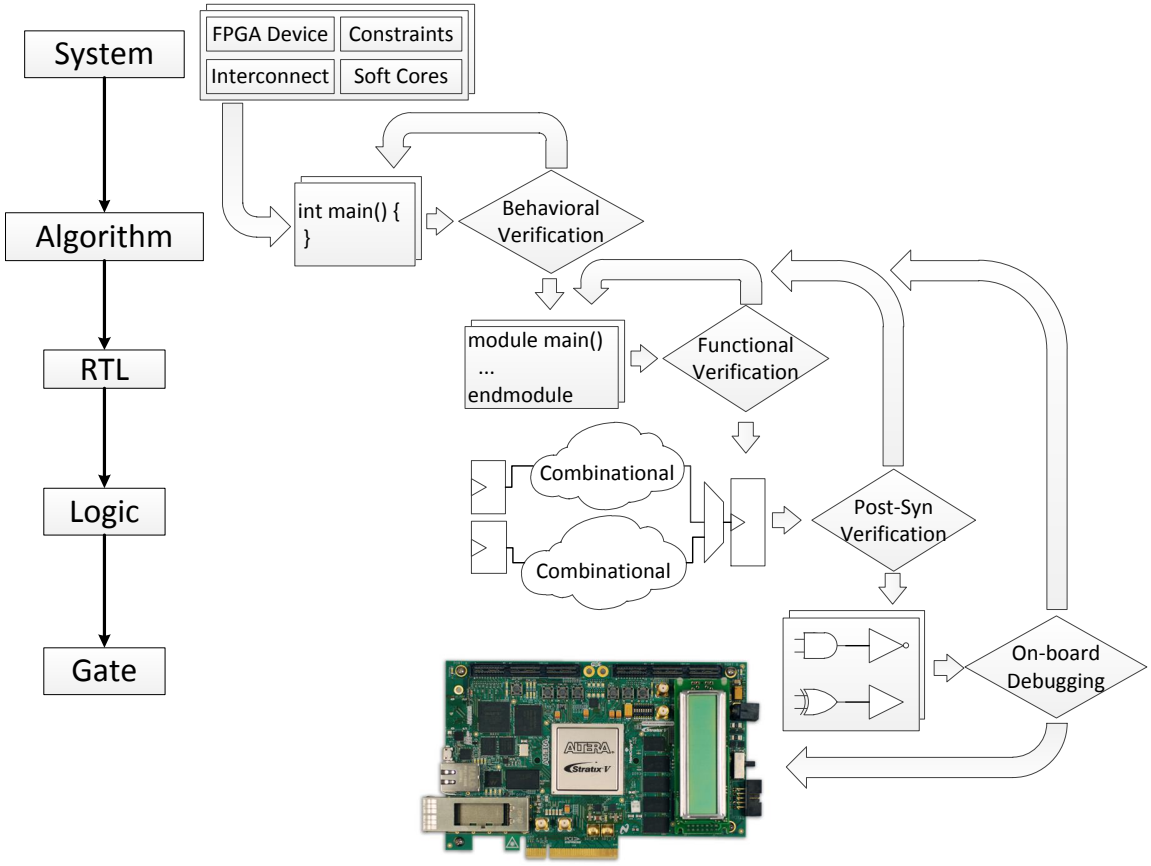

Figure 1.1: Hardware Design Flow, exemplified by Altera FPGA [1]

A general hardware design flow is illustrated in Figure 1.1, with the Altera Stratix V target platform [1] as an example. Hardware design starts with overall high-level

specification of the system-level architecture including architectural components such as platform, interconnect fabric, and microprocessor components such as NiosII [4], MicroBlaze [5], or ARM Cortex-A [6] cores. This high-level specification may be based on selection of a specific FPGA-based implementation platform, or high-level modeling of components through system-level design and virtual platform modeling [7, 8].

In parallel with system-level specifications, domain experts generate algorithm specifications typically as a behavioral description in a high level language (HLL) such as C/C++. The algorithm description is untimed and free of specific hardware structure; this specification is used for early functional verification in software, but the description does not contain information about the timing (e.g. latency) or implementation of computation or communication components.

With the architecture specifications and algorithm specifications, the designer performs hardware/software (HW/SW) partitioning to determine how algorithm components are mapped to the desired architecture. At this stage, designers perform high-level modeling to determine whether constraints in area, latency, throughput and power-consumption are feasible, and iteratively modify the architecture, algorithm or both until the constraints are estimated to be feasible. Because the platform characteristics are only estimates, there may be a need for further iteration after detailed implementation, which can significantly impact design flow productivity. Thus, the fidelity of high-level estimates to feasible implementations is important to overall productivity.

Given an algorithm and HW/SW partitioning, the untimed behavioral description is then translated into register-transfer level (RTL) implementations using hardware description languages (HDLs) such as Verilog and VHDL. The RTL specifies controlling state machine(s), functional units, memories, and interconnect fabric, and implements the components, interconnect, and timing of operations. The translation to RTL may be a manual process by experienced designers or an automated process through high level synthesis (HLS). The RTL specification is verified in functional simulation by comparing to expected results from the algorithm specifications.

After RTL implementation, computer-aided design (CAD) tools are used to synthesize the hardware into an implementation netlist, followed by technology mapping,

placement, and routing to realize the RTL on an implementation platform. The post-synthesis implementation is again verified through detailed timing simulation, and the design will be verified again through post-fabrication testing in the case of ASICs or on-chip testing in the case of FPGA platforms.

As shown above, the overall hardware design flow is a comprehensive stack of steps in which various practices like specification, partitioning, design, verification and integration are involved in different stages. To achieve a good quality design, each step requires extensive, detailed effort, and the steps may potentially be iterative to refine the design to meet goals in area, latency, throughput, achievable frequency, or power. Thus, the overall process requires significant effort, and the productivity of the flow requires effective methods for each step, integration between steps, and minimization of time-consuming design iterations.

This significant design effort is critical to both QoR and the non-recurring engineering (NRE) costs in labor and time to market. The competition among hardware companies is continuously increasing, and time-to-market is more critical than ever to the success of the deliverable products. Time-to-market directly impacts the addressable market, revenue opportunity and thus potential profit. Because the productivity of the design flow has direct impact on the required design cost and the time-to-market of products, it becomes a critical factor for the SoC design cycle and the success of the deliverable products.

Similarly, the underlying technology for IC fabrication keeps improving the feature size with recent ICs such as the Intel i7 Skylake and Altera Stratix 10 with 14 nm feature size. This advancement leads to significant improvement of IC capacity in transistors, which allows implementation of much more complex designs in a single chip. Hardware designers attempt to leverage this capacity to create more powerful and complex designs for improved functions and innovative applications. However, this trend imposes higher requirements for design productivity due to increased design complexity. As the complexity continues to escalate, it becomes harder for designers to effectively implement larger and more complex designs with traditional design flows; human productivity has been consistently falling behind the productivity requirements for complex designs. Figure 1.2 (adapted from [2]) illustrates that the gap

between the required and achievable productivity has been growing.

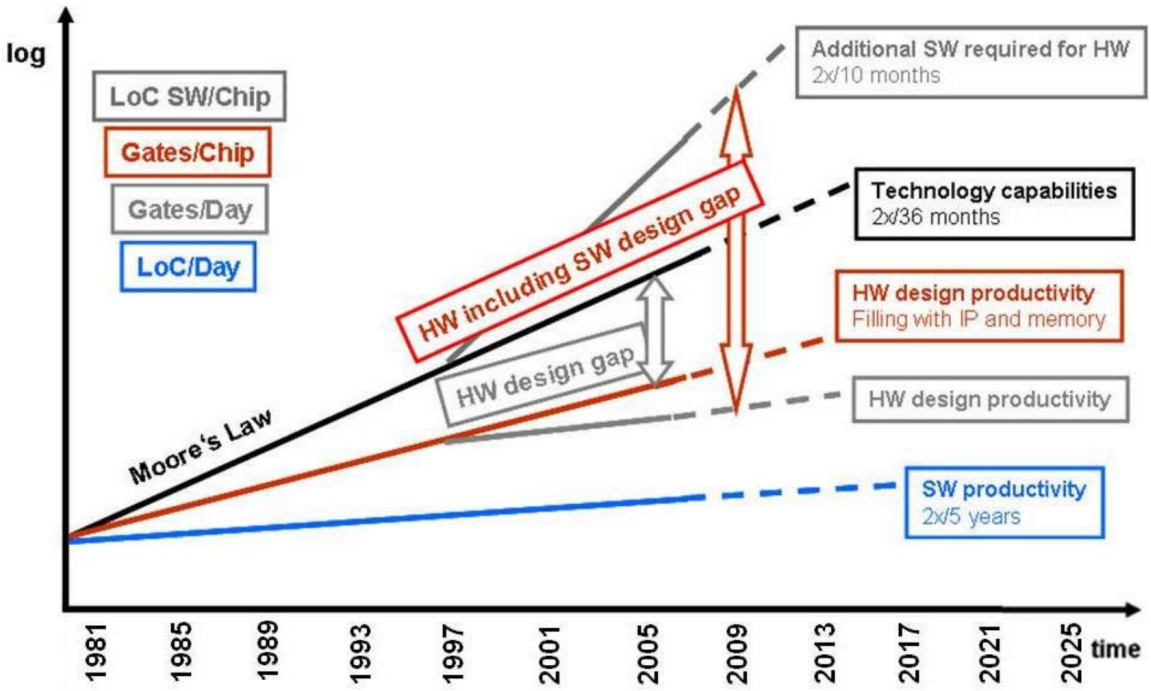

Figure 1.2: Hardware and Software Design Gaps versus Time [2]

In order to bridge that productivity gap, the design flow has also been evolving to catch up with the capability of managing increased capacity and design complexity. As the key to improved productivity, raising the level of design abstraction enables adoption of a design methodology which produces high QoR with less effort in detailed implementation, better decision-making in early, high-level design stages, and both fewer design iterations and lowered design effort to iteratively refine. From the correspondence between the levels of abstraction and the steps in hardware design flow shown in Figure 1.1, we can see that the hardware design flow is a top-down design process, in which the specification in each level is lowered down to the next lower level for more detailed implementation. Lower levels of abstraction require considerably more implementation details, and thus designing in a lower abstraction level not only imposes tedious and time-consuming implementation procedures, but also considerably more expertise for handling those underlying implementation and verification details. As electronic design automation (EDA) tools improve, it has been

observed that many low-level implementation details can be handled automatically by tools with equivalent or even superior quality, thus offering increased productivity by reducing human design effort and hiding common, easily automated details from designers.

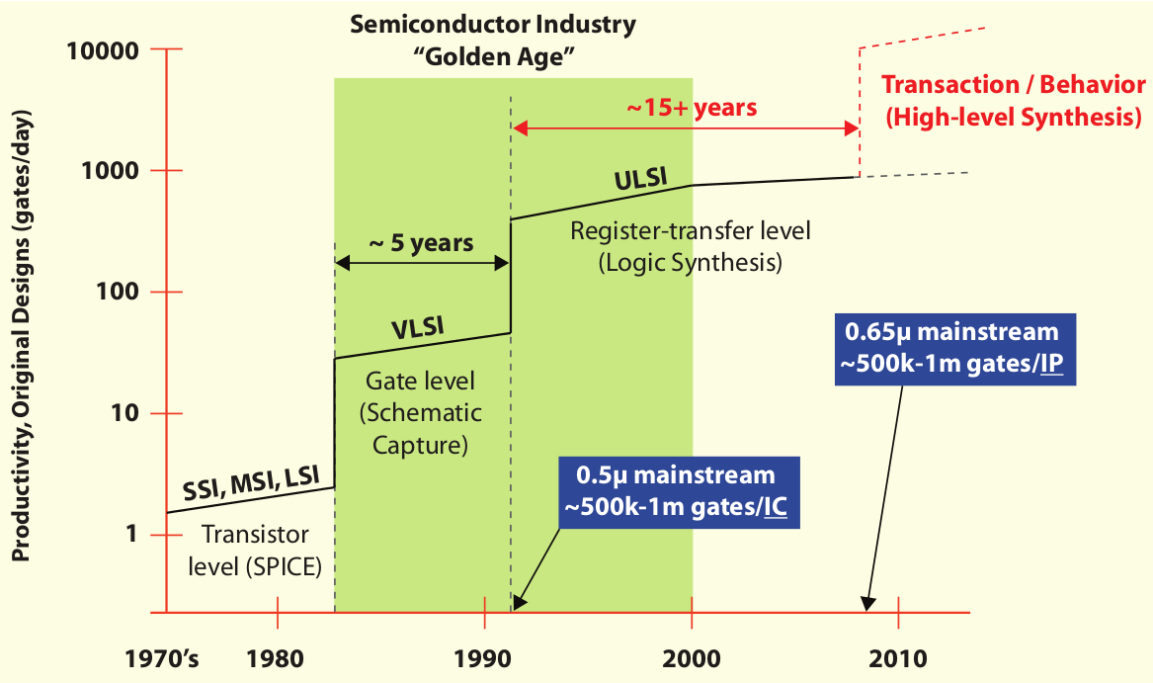

Figure 1.3: Productivity versus Design Methods, from [3]

Figure 1.3 from [3] illustrates the relationship between productivity and the design methodology abstraction level. It shows that the escalations of abstraction level, such as from transistor level to gate level in 1980s and from gate level to RTL in 1990s, have yielded significant productivity improvement. This demonstrates the potential of improved productivity through design at higher abstraction levels, but also demonstrates that productivity improvements have stalled since the emergence of RTL design. Given this context, under the ever-increasing pressure of managing rapidly escalated design complexity for modern applications, significant research effort has been spent searching for new design methodologies beyond RTL that increase the abstraction level and improve productivity. These efforts include coarse-grained

reconfigurable architectures (CGRAs), overlay architectures, multiprocessor system-on-chips (MPSoCs) and HLS (advocated in Figure 1.3), to obtain that required productivity. These methodologies are reviewed as follows in terms of potential productivity improvement achievable from the characteristics in their design flows, yet this thesis is targeted at the productivity improvement of HLS methodology.

1.1.1 CGRA

CGRAs consist of an array of functional units (FUs) interconnected by a mesh style network [9], with distributed register files to hold temporary values. This array of FUs provides computational power that is capable of doing byte- or word-level computations efficiently [10]. From the perspective of design methodology, in contrast to fine-grained bit-level FPGAs, CGRAs feature shorter synthesis and reconfiguration time by allowing easier mapping to the coarse-grained blocks with less configuration data compared to fine-grained FPGAs. CGRAs trade design space flexibility for improved productivity in synthesis, and reconfiguration time compared to fine-grained design methodologies.

As shown in [11], CGRAs mainly focus on the efficient execution of loops by accelerating the inner loop bodies. Many scheduling techniques for CGRAs require that loops need to be free of control flow transfers and conditional statements need to be if-converted [12, 13, 14]. CGRAs may also require a general purpose CPU for execution of other parts of the application. The design of CGRAs is domain-specific in that the parameters used in the design, e.g. fixed-/floating- point functional units, bit-widths of operations and the word length of configuration, vary greatly from case to case. Therefore, as illustrated in [11], despite the extra effort involved, a tuning process can be helpful for achieving better performance improvement. In the case for domain-specific applications, the tuning process with experimentation with various loop shapes and underlying compiler optimizations can also be reused in that domain for generic productivity improvement.

1.1.2 Overlay

Overlay architectures are specialized, often domain-specific designs for FPGAs that can in turn be configured to implement a variety of applications. Overlay architectures are virtual FPGAs that offer the configurability of FPGAs with reduced synthesis/compilation time, smaller configuration bitstreams, application portability, improved design reuse, and improved programmability [15, 16, 17, 18, 19, 20], thus leading to improved productivity.

The key concept of an overlay architecture is a virtual coarse-grained layer, which consists of necessary resources for mathematical or logical operations and routing, that enables easier management of a large amount of resources of FPGA platforms in an efficient way. The overlay architecture can be designed once for a domain of applications and reused many times by different applications within the domain, which significantly improves productivity compared to from-scratch hardware design for each application. This reusability considerably reduces synthesis time, and automated mapping of compute kernels in HLL code to the overlay architecture provides rapid compilation time for faster exploration of high-level kernel descriptions and overlay architectures. All of the characteristics above improve productivity by both facilitating the utilization of rich logic resources and speeding up the design iterations and exploration for performance tuning.

In overlay-centric design, the RTL design of the overlay is produced manually, with domain-specific customizations. For example, the array of island-style tile architectures in [19], where each tile contains a functional unit (FU) built from a processing element (PE) consisting of a DSP block and shift register LUTs, switch and connection boxes, is proposed for faster compilation time with throughput optimization; while the cone-shaped cluster of FUs in the overlay architecture proposed in [20] targets the reduction of area overhead for compute-intensive kernels.

With the RTL overlay, designers can implement a variety of applications with similar features by compiling application specifications to the overlay. Synthesis and mapping to the RTL overlay is typically faster than general FPGA synthesis, offering an improvement in synthesis productivity. Thus, overlay-centric design targets overall productivity improvement by reusing a good RTL overlay architecture instead of

performing detailed FPGA synthesis, which reuses domain-specific optimizations and reduces synthesis and design cycle time.

1.1.3 MPSoC

In addition to CGRAs and overlay architectures, FPGA-based MPSoC systems, which contain multiple processors, such as ARM Cortex cores, and programmable logic resources, also promise improved productivity. MPSoC systems, such as Zynq UltraScale+ MPSoC [21] from Xilinx, feature the integration of the software programmability of a processor with the hardware programmability of an FPGA, thus leading to a more software-style development process for the design of hardware platform. Some MPSoC platforms use the entire FPGA as a network of interconnect CPU cores [22, 23, 24]. These platforms target improved overall productivity by transforming the FPGA platform into a software-programmable multiprocessor.

Like overlays, MPSoCs spend significant effort on designing a single good implementation that can be reused by many applications. However, MPSoCs are designed for software programmability instead of the virtual FPGA of overlays. MPSoCs sacrifice general purpose customizability for a fully software programmable platform; there is limited customization for individual applications, but easy mapping of software. Although MPSoCs allow customization of individual CPU cores, it is generally limited to specialized instructions rather than full application accelerators.

1.1.4 High Level Synthesis (HLS)

HLS offers improved productivity by automating transformation of high-level, untimed- or partially-timed specifications to low-level cycle-accurate RTL specifications. Because HLS methodology accepts more concise algorithmic descriptions in a higher abstraction level, high level languages (HLLs), it features easier behavioral testing and significant reduced expertise requirements with underlying hardware. HLS has achieved significantly improved productivity with reduced time-to-market and is increasingly the design entry method of choice for hardware design. Based on comparisons of HLS and RTL design methodologies, Meredith [3] found that HLS is

3× more efficient/compact in terms of lines of code, HLS has $5-10\times$ faster simulation time in SystemC compared to RTL, and thus HLS is more than 10× more productive overall.

However, despite the productivity improvements of HLS, the overall productivity of the design flow remains a challenge as the complexity of applications continues to grow. Hence, the need for improved productivity to handle the explosion of capacity and design complexity is stronger than ever. In order to comprehensively understand why the current strategies are insufficient for productivity improvement, and identify opportunities for further improved productivity, the design flow is examined as follows.

The HLS design flow that maps HLL code to RTL implementations is shown in Figure 1.4. We divide the flow into three stages: HLL compilation, intermediate representation (IR) to RTL transformation, and RTL verification and synthesis. The flow begins with the design input in high level specifications, such as C/C++ and SystemC, as the starting point of the first stage. The high level specifications are compiled into an IR using a compiler language front-end such as Clang [25] (built on LLVM [26]) to parse the input description and map to a target independent IR. Optimizations for compilation, such as dead-code elimination, constant propagation, strength reduction, and loop unrolling are optionally performed on the IR to optimize; industry tools such as Xilinx Vivado HLS [27] also use pragma-guided optimizations to transform input code in preparation for RTL optimizations such as pipelining.

The optimized IR is then passed to the IR to RTL transformation stage, which contains the main HLS algorithms of allocation, scheduling and binding. Allocation defines the types and quantities of hardware resources that will be used to implement the behavioral specifications. Scheduling determines the order of operations given data dependence requirements, resource limitations (as determined by allocation), and design constraints such as target frequency or latency. Finally, binding maps each operation to a specific functional unit given the schedule and allocation, while simultaneously attempting to minimize intermediate storage and multiplexing requirements induced by the mapping.

Allocation, scheduling and binding, are executed in different orders by HLS tools.

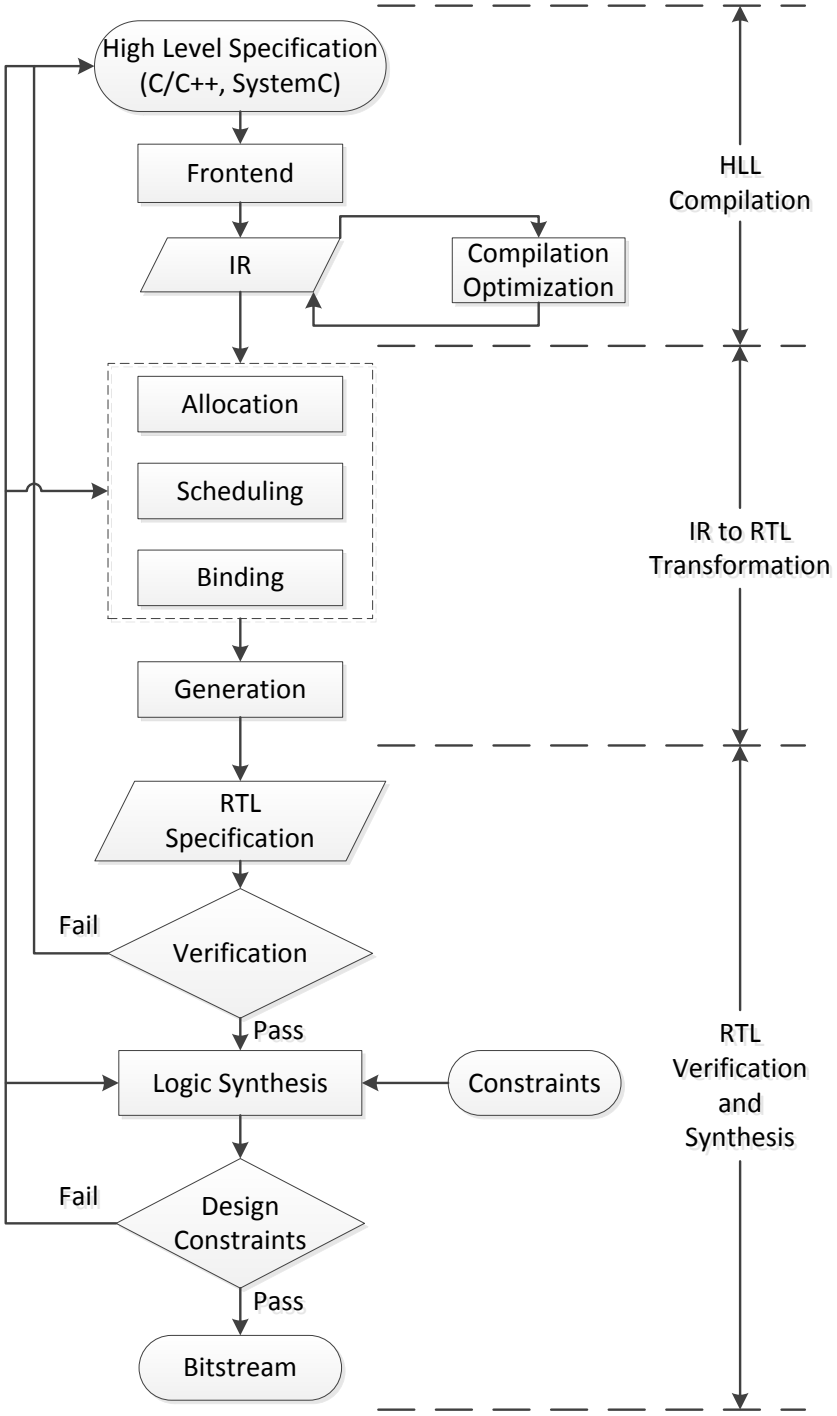

Figure 1.4: Design Flow with HLS

For example, the allocation of connectivity components, such as buses, can be added before or after binding and scheduling tasks [28]. For design space exploration, some HLS tools may apply an iterative mechanism to execute the tasks one after another iteratively for specific optimizations towards the design constraints. After allocation, scheduling, and binding are complete, we finally generate RTL specifications.

The compilation and IR to RTL stages usually only accept source input in HLL. However, designs may have over 70% reusable intellectual property (IP) blocks [29], which significantly affects design effort. Most IP blocks are provided as RTL descriptions, and thus it becomes a disadvantage for HLS tools that only accept source input in HLLs and cannot integrate those reusable RTL IP blocks. In order to achieve improved productivity by taking into account those RTL reusable IPs, HLS methodology need an efficient scheme of integration.

After RTL generation, we first perform simulation to verify the correctness of the design. Simulation of the RTL specifications may use ModelSim [30] or translation to SystemC and verification together with the high level specifications [31]. AutoPilot[1] [32] uses this approach to integrate the RTL simulation and the original C/C++ or SystemC specifications. The verification process may require iteration to correct high level specification errors before passing the RTL to synthesis tools such as Quartus II [33], or Vivado [34] for logic synthesis and implementation on the target platform.

As discussed in Section 1.1, verification effort is increasingly critical, with over 50% of the overall project time now spent in verification on average [35]. As such, the verification process of hardware design has become the bottleneck for achieving improved productivity. Furthermore, because HLS-produced RTL is not human-readable, and the debugging is particularly time-consuming and cumbersome, back-tracing through hundreds of signals and simulation cycles, HLS-based design verification is especially challenging. As a result, verification in HLS-based design flows is even more critical, exposing additional challenging verification issues due to HLS.

Logic synthesis accepts two inputs, HLS-generated RTL and design constraints. First, the RTL specifications are elaborated and technology-mapped to produce a

[1]Now part of Xilinx Vivado

circuit-level description. The circuit is then placed and routed (PAR) based both on the circuit structure and the design constraints. Logic synthesis produces detailed reports including resource usage and maximal achievable frequency. If constraints are not satisfied, designers have three options to refine the design. First, designers may examine the synthesis reports, refine constraints and attempt different synthesis settings. Next, designers can alter HLS options and iterate with the HLS process of allocation, scheduling and binding to attempt to reorganize the RTL structure in order to modify the critical path and thus achieve the design constraints. Finally, the HLL source code can also be refined to alter the algorithm structure. Once all constraints are met, the design is finalized and a bitstream is generated to program the target FPGA device.

Together, we identify that efficiency in use of reusable IP blocks and the verification and debug process are two critical factors affecting productivity of HLS-based design flows. Various research efforts attempt to address aspects in productivity, including performance monitoring and prediction [36, 37] to reduce design iterations by improving early identification of bottlenecks, HW/SW co-design [38, 39, 40] to improve early mapping decisions, integration of reusable components [41, 42], and verification [43, 44, 45, 46, 47] for better debugging efficiency.

1.2 Motivation and Thesis Statement

Integration of reusable components and improvement of verification and debugging stand out as two of the most critical bottlenecks to improving productivity in HLS-based design flows. We now take a detailed look at these two factors to examine the challenges in HLS flows and how these challenges affect overall productivity. From the relation of these challenges to productivity, we motivate productivity improvement of HLS-based design flows through improving integration of reusable components and improving verification and debugging of HLS-produced designs. In this thesis, we emphasize productivity for FPGA platform targets, but ASIC platform targets are closely related.

Reusability in design is a critical factor to deliver improved productivity. As

systems become increasingly complex, sub-systems and components are repeatedly instantiated; reusing prior implementations of these components reduces design effort both for implementation and optimization of the components and allows designers to concentrate effort on design of new features rather than reimplementation of prior well-known hardware [48]. Design reuse has been long known as a general goal for improved productivity and QoR, and industry adoption of reuse in design flows is well documented [49]. Industry studies demonstrate trends reducing amount of new logic designed with corresponding increases in in-house and third-party IP reuse [48], with as much as 70% of designs on average contributed by IP-cores [29]. Despite this, the support for IP integration within HLS-based design flows is limited, which creates a significant productivity bottleneck for HLS-based design.

Despite the motivation of improving productivity by enhancing the reusability, the integration of RTL IPs in HLS methodology is challenging. In traditional RTL design, reusing RTL IP cores relies on manual instantiation and connection of ports, which requires detailed hardware expertise to decide how to instantiate, how many copies of an IP to instantiate, and how to implement required communications. HLS-based designs must support integration of IPs with automation to decide how many instantiations of an IP are required, implement communications between IPs and HLS-produced logic, share the IP blocks between multiple users, and analyze IP use patterns to ensure that each block is used effectively (maximizing performance). Furthermore, designers use a wide variety of highly optimized RTL IPs; HLS-based flows must easily support these different IPs to allow effective designer-guided exploration of which IPs are the best choice for a particular application.

Based on these challenges, we will present a detailed analysis of integration strategy and present a general solution for IP integration in HLS that supports a wide variety of IPs for behavioral-level IP integration in Chapter 4. We will present case studies that demonstrate how our IP integration technique can be used both to effectively integrate third-party IPs, but also to improve productivity in general design through effective integration with non-synthesizable IPs for functional simulation and debugging.

In addition to the problem of reusability in HLS-based hardware design flow,

functional verification and debugging is increasingly difficult and time-consuming. In hardware design, verification is the process that determines if the implementation of design passes the test during execution (either simulation or emulation) and meets the specification of functional requirements. The process of verification does not necessarily determine the location or root cause of a bug that caused the incorrectness and verification failure. Debugging is the detailed process that investigates, diagnoses and finds the root cause of a bug. As designs become more complex, functional verification and debugging is increasingly expensive, consuming over 50% of time on average [35]. Therefore, in recent years, there has been a proliferation of works that are proposed to address this verification problem [43, 45, 50, 46, 51, 52, 53, 54, 55]. However, debugging HLS-produced designs is especially challenging because HLS-produced RTL is not intended to be human-readable. Thus debugging machine-generated functionally incorrect RTL is time-consuming and cumbersome, requiring back-tracing through hundreds of signals and simulation cycles to determine the underlying error. Furthermore, the exposure of a bug is not necessarily the source of a bug, nor is the location of an RTL error easily correlated with the line of HLL code. This challenging process requires support in the HLS flow to enable fast and efficient detection of the bug source in the source code.

Development with HLS tools starts with a traditional software design flow: an algorithm is implemented and verified as functionally correct in software using tools such as GNU Project Debugger (GDB) or Valgrind [56] before passing to HLS for RTL generation. After the software stage, the source is free of *deterministic* bugs, but software execution can mask *non-deterministic* bugs that will lead to functionally incorrect hardware. For example, a *non-deterministic* random value that is introduced due to an uninitialized value or address in software can have a different random value or a fixed reset value in hardware because it is handled in a different manner in hardware than the undefined software behavior. Similarly, a *non-deterministic* access to a random value that is introduced due to an out-of-bound access of an array in software can also have a different random value or a rollback value in hardware because the hardware implementation of handling this *non-deterministic* access can be different

from the software implementation. In addition, subtle bugs activated due to *non-deterministic* packet arrival orders may be masked in software but produce incorrect hardware. These *non-deterministic* bugs that can lead to discrepancies between software behavior and hardware behavior are typically more challenging to identify; they may be data-dependent or produce no functional incorrectness in software yet produce incorrect hardware. Although it is possible to debug deterministic bugs as well, it is easier to find such bugs in traditional software tools, and thus users generally only perform RTL debug when software tools are insufficient.

HLS tools assist through testbench generation and integration with software testbenches [34, 57, 58, 59]. Many academic efforts also assist in automating signal selection and efficient use of trace-buffers for emulation-based verification and debug [53, 54, 55]. Furthermore, recent works have used automated comparison of software and hardware execution to help determine the earliest instance of execution mismatch [60, 50]. However, the first detected execution mismatch is not necessarily the source of a bug; for example, we may detect an execution mismatch at an array out-of-bounds access, but the source is the instruction(s) that set the index variable incorrectly – in a complex, non-human-readable datapath, it is important that HLS tools assist in tracing a bug to its source. Based on the motivation and challenges in the verification of HLS-produced designs, we examine the verification process in detail in Chapter 5 and demonstrate an automated cross-layer verification framework that efficiently facilitates the verification process by precisely localizing the root-cause of source-level bugs, thus significantly improving the efficiency of debugging and the overall productivity.

In addition to verification of HLS-produced designs, verification and debugging of HLS tools have an indirect yet critical impact on productivity of HLS-based design flows. The correct implementation in an FPGA hardware platform for a behavioral specification relies on not only the functional correctness of the source input, but also the correctness of the internal transformations and optimizations within the HLS tool. Thus, in addition to the debugging of source input, the verification and debugging productivity for the correctness of the HLS optimizations complementarily have non-trivial impact on the verifiability as well as overall debugging productivity.

HLS-supported optimizations are critical to achieving high QoR, and designers must continue to iteratively refine HLL source and use of reusable components until they meet their design objectives. Thus, HLS tools with more (and higher quality) optimizations will allow designers to more quickly meet design objectives and improve overall productivity. However, advanced optimizations are challenging to implement and deploy – and challenges in verifying the correctness of optimizations can delay delivery of key optimization advances.

HLS tool development is a huge software development effort and a significant portion of that effort is spent on verification. Traditionally, software verification uses debugging tools (e.g. GDB) and memory analysis tools (e.g. Valgrind [56]) in the development process. However, HLS tools are more difficult to verify compared to typical large-scale software systems; HLS tool output must be subsequently verified through functional verification of the generated RTL implementation. Since development of optimizations in HLS involves many complex transformations, verifiable correctness of output is further exacerbated. Thus, there is a need for improving the verifiability of the functional correctness of HLS tools to confidently deliver quality design along with the improved productivity.

To address this need, many efforts have been conducted to find appropriate approaches for guaranteeing functional equivalence between the input specifications in HLL descriptions and output design in RTL descriptions to verify the correctness of HLS tools [61, 62, 63, 64]. The formal methods used by these works statically analyze the behaviors of models described in both HLL and RTL descriptions, and perform the equivalence verification by using techniques like sequential equivalence checking (SEC) in [61] and bounded model checking (BMC) in [64]. However, formal verification usually relies on solving boolean satisfiability (SAT) with tools such as Chaff [65], and processing time increases exponentially with the design size. This increase of processing time results in impractical formal verification proofs for large and complex designs, thus significantly increasing the difficulty of equivalence checking between the input and output descriptions of the tool.

Verification of an HLS tool flow is also similar in concept to verification of general purpose compilers, which is a known and challenging area of study. Typical

compilation frameworks such as LLVM [26] are not formally verified; some projects attempt to verify LLVM passes, but have not successfully produced a fully-verified compiler [66]. In contrast, projects such as CompCert [67] build compilation frameworks in Coq [68], a proof assistant language, so that the resulting compilation framework can be formally verified, but these frameworks typically lag significantly in QoR due to the challenges in developing advanced optimizations in formally verifiable proof languages.

Therefore, instead of proving that the HLS tool (compiler) executes to completion correctly, we must prove that output RTL (or binary) is functionally correct. Since the HLS framework is normally built on general compiler infrastructure, such as LLVM, we cannot rely on underlying formal verification to ensure that all compilation transformations are provably correct. Some prior work has tried to formally verify functional equivalence between input high level sources and the output RTL [61, 62, 63]. However, the complex control data flow graph (CDFG) transformations and hardware-oriented optimizations prove difficult to formally verify: many valid, important performance optimizations are rejected because the formal proof cannot verify functional equivalence. As the complexity of HLS optimizations increases, this problem will only be exacerbated, especially in a framework that has no underlying formal framework.

Compared to those static formal methods, dynamic methods verify the functional correctness via simulation. Since the simulation uses real test vectors to verify the functionality, the complex transformations and optimizations rejected by formal methods can be accepted in the test flow. Furthermore, without the SAT solving which grows exponentially with design size, verification time can remain feasible even for large and complex designs given reasonable test vectors. However, as noted by [61], traditional simulation-based approaches may only find the discrepancy in the behaviors of HLL descriptions and RTL descriptions at the end of the simulation, which may be millions of cycles after the bug is introduced. The large detection latency of a bug not only increases the verification processing time significantly, but also imposes significant effort on designers and verification engineers to correctly select and monitor a large set of relevant signals and backtrace them for millions of cycles.

As capacity and complexity of designs grow, this traditional simulation-based method becomes impractical for large designs. Therefore, it is critical for dynamic methods to overcome these challenges to deliver both useful diagnostic information and reduced simulation time.

Complementary work in improving dynamic methods of tool verification include automated test case generation [69] by generating random programs with user-specified features to verify correctness of HLS tools through random test generation and verification. Additionally, some post-silicon verification tools provide some information to diagnose that the source of a bug is due to the HLS tools rather than HLL source [70]. Although it helps HLS developers to automatically verify optimizations, they do not address challenges in providing useful diagnostic information to the relevant bugs and reducing the simulation time of the verification effort. Both works concentrate on determining the existence of a bug rather than diagnosing the source of the bug or assisting in the debug process.

In Chapter 6, we present a trace-based verification technique that automatically inserts verification code into the generated RTL to assist in debugging of HLS tools through detailed analysis of both the existence and potential sources of RTL bugs due to HLS tool errors. Furthermore, we present several real case studies of how our verification infrastructure aided in the debug process to diagnose and solve rare bug conditions in our HLS tool. The objective of this work is different from that of the work presented in Chapter 5 in that this work is addressing the debugging of HLS tools to improve the debugging productivity during the development of HLS optimizations, while the work in Chapter 5 is addressing the debugging of the source input. These two different objectives complementarily address two important aspects which together contribute to the productivity improvement in the overall verifiability.

1.3 Contributions and Organization

Based on the observation of the critical need for improved productivity, this thesis specifically targets improvements in reusability and verification of both user's source and the HLS tools. Specifically, this thesis contributes to productivity in these areas

with:

- **Improved reusability through behavioral IP integration.** To improve productivity through enhanced reusability, we demonstrate behavioral-level IP integration that allows integration of RTL IPs automatically by the HLS tool. Behavioral-level IP integration allows user-specified mapping of high-level language functions (or instructions) to RTL IPs, with automated integration of those IPs including multiple instantiation and sharing of the IPs among multiple users. Our behavioral-level integration supports both synthesizable and non-synthesizable IPs as well as both fixed- and variable-latency IPs.

- **More efficient debugging of HLS-produced design.** As the productivity of the design stage improves through HLS-based flows [71], productivity in verification of HLS-produced designs is critical. We demonstrate an automated framework that instruments user source code, performs discrepancy analysis comparing high-level behavior to RTL behavior, and when discrepancies are detected, automatically traces the discrepancy to the bug activation as well as the bug source in high-level language source. Through this automated framework, we substantially improve the productivity of verification and debugging by not only bridging the complexities in debugging machine-generated RTL, but also by automating tracing and identification of bugs in functionally incorrect simulations.

- **More efficient functional verification of HLS tools.** In order to improve developer ability to deploy advanced HLS optimizations, it is critical to provide effective development tools for testing, verification, and debugging of the HLS tool. Improved HLS optimizations indirectly improve productivity through reduced user effort to achieve design goals, fewer design iterations, and improved ability to achieve high performance with minimal high level source modifications [72]. We demonstrate an automated instrumentation framework that assists in verification and debugging of HLS tools through detailed verification of generated RTL, and tracing of RTL bugs to underlying causes. We demonstrate how this framework has been used to find a variety of rare corner-case bugs in

our HLS tool, aiding verification and debug and thus helping to improve HLS QoR.

This thesis is organized as follows. In Chapter 3, we present a discussion of related works in productivity improvement. Chapter 4 develops a generic behavioral-level IP integration framework for HLS that supports fixed- and variable-latency, synthesizable and non-synthesizable IPs without requiring application partitioning. It also presents four case studies to demonstrate flexibility, utility and productivity improvement. Chapter 5 presents AutoSLIDE, a cross-layer verification framework that automatically instruments critical operations, detects discrepancies between software and hardware execution, and traces the datapath to precisely pinpoint the root-cause of source-level bugs, thus substantially reducing user effort to localize bugs and enhancing debugging efficiency for productivity improvement. Chapter 6 presents an trace-based verification technique to automatically insert verification code into the generated RTL to assist in the verification of HLS tools. By facilitating the development of HLS optimizations with better debugging efficiency, this technique indirectly speed up the delivery of better QoR of HLS-produced designs, thus significantly reducing the effort and time in the design iterations for refining the QoR and implicitly improving the overall productivity. Chapter 7 summarizes the thesis and highlights the future work and potential enhancements for further improved productivity in the overall hardware design flow.

Chapter 2

Related Work

As a promising design methodology for improving productivity, HLS has been increasingly adopted due to the increasing challenges of handling more complex designs in RTL. Based on the discussion in Chapter 1, there are challenging productivity bottlenecks including IP integration for behavioral level reusability, source-level debugging of HLS-produced designs and functional verification of HLS tools. Significant research effort has been spent on addressing these problems.

In this chapter, we examine both research works proposed directly for productivity improvement and works proposed for other objectives such as performance improvement that indirectly improve productivity. First, we investigate prior related work in IP reuse and the verification process in the design flow using HLS. Within reusability, we categorize works into system-level and behavioral-level IP integration. Verification works are also classified into groups based on source-level debugging, simulation-based or in-situ approaches, as well as HLS-tool verification. Finally, we also discuss a variety of HLS research targeting general QoR improvement with an indirect impact on productivity, including front-end compilation, scheduling, allocation, binding, and miscellaneous optimizations of HLS algorithms.

2.1 Reusability

In electronic design, an IP core is a reusable unit of logic, cell or chip layout design. These IP cores can be used as building blocks for ASIC chip design or FPGA-based logic design, thus are important for fast and efficient hardware development [73].

In order to deliver productivity and performance gains to handle the increasing complexity of electronic designs, IPs are often used because they are pre-designed, optimized and pre-verified. In HLS-based design, IP integration can be performed at the system level or behavioral level. In system-level IP integration, RTL IPs (either HLS-generated or user-specified) are manually instantiated and connected after HLS. HLS-generated RTL IPs are entered into an IP library with other system-level components, which may include generation of protocol interfaces for the IPs to ensure communication compatibility. Then, a system-level RTL module or schematic is designed by manually instantiating and connecting both user-provided and HLS-generated RTL components.

In contrast, behavioral-level IP integration directly instantiates the RTL IPs during HLS by mapping HLL functions or instructions to IP uses. By directly integrating RTL IPs, HLS tools automatically handle instantiation of IPs, sharing of multiple IP uses, and connecting IPs to other components, leading to significant savings in human effort and thus overall productivity. In this sub-section, we discuss the two groups of works, which are related to system-level IP integration and behavioral-level IP integration, respectively.

2.1.1 System-Level IP Integration

System-level IP integration features post-HLS instantiation of IPs and manual connection with other system-level components. Prior EDA tools for high-level simulation and architectural exploration provide IP integration using a generic, high-level interface to simulate IP behavior in system-level designs through transaction level modeling. However, as high-level exploration tools, they do not generate synthesizable RTL specifications for the architectures. Instead, they expect designers to

realize system-level designs through manual, detailed RTL implementation. Coware[1] N2C [74] provides a virtual bus interface to connect modules and third-party IPs for architecture co-simulation. Cadence proposed Virtual Component Co-design (VCC) [75] for system-level co-design and IP reuse, but does not synthesize IP interfaces. These tools provide a framework for high-level system evaluation, but do not implement required details to generate functional RTL implementations.

In order to accelerate system-level IP integration with functional RTL, leading FPGA vendors also provide system-level IP integration tools. Altera provides a GUI-based integration tool, Qsys [76] (replacing previous SOPC builder [77]) and Xilinx provides Vivado IP Integrator [78], both targeting the acceleration of IP integration. These tools accept user-provided or HLS-generated IPs and assist in integration through automatic generation of interconnect logic, such as address/data bus connections and arbitration logic, and standardized interfaces like Avalon [79] and AXI [80], so that the manual effort of connecting interfaces and creating controlling state machines is reduced. These tools accept HLS-generated RTL IPs [27, 81, 82, 59], leading to an efficient tool chain that increases the efficiency of integrating HLS-produced IPs. However, system-level tools do not generalize IP integration; the user must still manually partition C/C++ code into portions that do not use IPs or re-implement non-synthesizable functions for system-level IP integration. Furthermore, system-level IP integration prevents design space exploration including multiple instantiation of IPs and modifying the HLS schedule to leverage IP use parallelism.

In summary, although system-level tools improve IP integration by automating the generation of interconnect logic and simplifying connection via GUI-based integration environments, they leave several major challenges for designers. Instantiation and connection of IPs with other system-level components, and partitioning of the application into HLS-compatible portions both require substantial design effort. Due to the extra design effort, the productivity of HLS is considerably underutilized, which motivates the need for more efficient IP integration strategies.

[1] Coware is now part of Synopsys

2.1.2 Behavioral-Level IP Integration

Behavioral-level IP integration automatically handles instantiation and connection of IPs, with direct mapping of IP use to instructions or functions and thus support for analysis of IP use during allocation, scheduling and binding during the HLS process. LegUp [83], an academic open source HLS tool, allows some IP integration for a fixed set of IPs. Floating-point operations in the HLL programs are supported by being implemented in the IP cores provided in megafunctions [84]. However, because of the hard mapping from specified operations to proprietary IP implementations, only megafunctions related to floating-point operations, such as *altfp_add_sub*, *altfp_mul*, *altfp_div* and *altfp_compare* are utilized targeting floating-point arithmetic. The other IP cores in megafunctions provided, such as *altfp_exp*, *altfp_log* and *altfp_matrix_mult*, cannot be taken advantage of for performance improvement. Similarly, Xilinx Vivado HLS [27] supports a small, fixed set of IPs for behavioral-level integration, but do not target generalized integration of third-party IPs. For example, trigonometric functions such as *sin()*, *cos()* and *tan()* or other common functions such as *sqrt()* can be implemented directly in CORDIC IP cores [85]. These two examples demonstrate IP integration and improved productivity through integration of library functions. However, because these methods do not target generalized integration of third-party IPs, they do not allow user management of operation or function-to-IP mappings. Thus, users can only integrate the fixed set of IPs defined by tool providers, which significantly restricts reusability to a predefined set of tool-provided IPs. In addition, there are non-synthesizable library functions, such as those from stdlib.h and math.h, which are heavily used in HLL applications to facilitate the development and verification of algorithms. Without the support for these non-synthesizable functions, the HLL applications using those library functions cannot be directly accepted by the HLS tool. This results in not only the non-trivial effort for partitioning the applications into synthesizable portion and non-synthesizable portion, but also the HLS flow not being able to leverage the existing implementations of library functions for efficient development and verification.

An alternate approach to behavioral-level IP integration translates RTL IPs into equivalent synthesizable C++ code [41]. Then, instead of instantiating RTL IPs, the

equivalent C++ code is re-synthesized by HLS. Although this provides an integration flow for HLS, the results of this method [41] show that it introduces area overhead, extra time to translate RTL IPs to C++, and limitations on the size and complexity of RTL input that can be supported.

Another academic approach [42] proposes a method for automated integration of IP accelerators from a user-provided IP library. Instead of a HLS-based method, this approach targets heterogeneous multiprocessors; users specify a function in high-level C/C++ code that will instead use the IP accelerator core. Then, their ESPAM [86] tool generates a system that connects CPU and IP cores, and generates a C/C++ program for each processor that uses the IP appropriately. This method only targets the design of a multiprocessor platform and relies on a standard C/C++ compiler (GCC [87] in this case) for the processors. Thus, despite the extension of IP sources integrated in HLL programs, it is not an HLS-based method for general hardware design in FPGAs or ASICs.

Another work [88] also proposes a method that tries to address the integration of IPs for basic matrix operations in HLS methodology. It identifies linear algebra matrix operations to be implemented by IPs, particularly matrix-matrix multiplication, matrix-vector multiplication, and matrix scaling. The operations are identified through pattern-based matching of directed acyclic graphs (DAGs) and use of vendor-provided IPs [89]. Although this technique is attractive for finding computation subgraphs implemented by IPs, it requires graph/sub-graph isomorphism to identify and match candidate graphs. Thus, to generalize to complex applications such as Black-Scholes [90] in computational finance and Secure Hash Algorithm (SHA) [91] in cryptography the mapping problem becomes more complex, which presents a scalability problem because the algorithm challenges scale with the size and complexity of the IP used.

Despite the improvements from prior works in IP integration, we identify several remaining bottlenecks to productivity that we will address in Chapter 4. Since our original work was published, Xilinx has also released SDSoC [92], which has a similar feature set, including user-specifiable IP support and behavioral-level IP handling.

- **Post-HLS system-level IP integration requires significant effort.**

System-level IP integration still relies on manual instantiation and connection of IPs, as well as partitioning HLL applications into functions that do and do not use IPs and reimplementing non-synthesizable functions. Furthermore, hardware design expertise is still required for the manual process, which imposes effort on users for design space exploration. To improve automation in the HLS-flow, we must integrate IPs at the behavioral level and modify compilation, allocation, scheduling and binding decisions to consider IP reuse.

- **Generalized, user-specifiable IP mapping in behavioral-level integration.** Prior behavioral-level integration works only support integration of a few basic mathematical functions, such as *sqrt()*, *div()*, and *abs()*, with fixed mapping to a proprietary IP library, such as LogiCORE cores [93] or megafunctions [84]. These constraints result in not only limited number of functions implemented by IPs, but also fixed implementations for the specified functions. Therefore, a generalized, user-specifiable integration method is needed for utilizing a larger range of IPs to achieve further productivity improvement.

- **Non-synthesizable HLL functions are needed for early evaluation and validation.** HLL applications make heavy use of non-synthesizable functions from libraries (e.g. stdlib.h and math.h) to facilitate the development and verification of algorithms. However, in HLS-based design flows, the non-synthesizable portion of the input code needs to be separated from the synthesizable part for HLS processing. This partitioning not only involves considerable human effort, but also eliminates the efficiency of using non-synthesizable functions in early evaluation and validation. Thus, there is a need to support non-synthesizable functions in IP integration for efficiency improvement.

In order to address these issues, in Chapter 4, we propose our behavioral-level IP integration in HLS [94] to efficiently support the integration of various user-provided IPs, such as fixed- and variable-latency IPs as well as synthesizable and non-synthesizable IPs, without requiring application partitioning for handling HLS-compatible and HLS-incompatible portions respectively.

2.2 Verifiability

As discussed in the motivation in Chapter 1, functional verification is another significant bottleneck to improved productivity. Significant prior research [43, 45, 50, 46, 51, 52, 53, 54, 55] has been proposed to address productivity of verification. We first discuss works directly targeted for improved verification productivity through source-level debugging, then discuss works that indirectly affect productivity by verifying HLS tools and supporting development of advanced HLS optimizations.

2.2.1 Source-Level Debugging

To directly affect productivity of source-level debugging, several academic and commercial HLS tools have added verification and debug support to their tool flows [47, 34, 58, 59, 43, 45, 44, 46, 95]. These verification techniques include detecting HW/SW execution discrepancy (simulation-based or on-board execution-based), developing software-like debug environments, and synthesizing C assertions to hardware.

Simulation-Based Discrepancy

Several commercial [34, 58, 59] and academic tools [47] couple hardware execution with a software reference model to compare outputs from both executions at runtime. However, they typically compare output streams at the interface level and thus lack detailed internal behavior to diagnose the cause for mismatched execution.

Campbell [95] proposes an automated cross-layer HW/SW co-simulation framework for faster bug detection. This work leverages HLS tools to map any synthesizable C/C++ function to RTL hardware, then uses Verilator [96] to translate the RTL hardware to an equivalent cycle-accurate SystemC code, and integrate the remaining C/C++ application source with the SystemC code to perform co-simulation. The execution trace collected from the co-simulation is compared with that collected from the software execution to detect the mismatch, which is used to backtrace to the location of the bug in the source. SystemC co-simulation achieves 10× speedup in simulation time, and automated of cross-layer translation, co-simulation and trace recording, accelerates verification with faster bug detection, both of which contribute

to improved productivity. However, after a bug is detected, localization of the bug source (i.e. the root cause of the bug), which is the challenging part of debugging [35], is still left to the designer or verification engineer.

Execution-Based Discrepancy

In addition to simulation-based discrepancy detection, on-board execution-based discrepancy detection facilitates debugging HLS-generated circuits by running and instrumenting the design *in-situ* on FPGA platforms, thus achieving full speed execution and debugging including system interactions [44, 46, 53, 54]. Based on the academic HLS tool LegUp [83], Goeders [44] proposes a method that inserts trace buffers to record the control and data flow in real-time execution, allowing the user to retrieve data and replay execution. This work also features a software-like debugging context that allows stepping and breakpointing of execution, which will be discussed later in the following sub-section. These approaches require trace buffers to record information for the following diagnosis of debugging using a *record and replay model* and efficient trace gathering. Therefore, the capability of tracing as much relevant information as possible, and recording for as long as possible is critical to overall verification efficiency. Therefore, Goeders [46] proposes a method to improve tracing efficiency by dynamically selecting relevant signals to record, thus allowing longer trace recording and enhanced visibility to find bugs more quickly. Yang [53] also targets efficient usage of trace buffers for silicon debug by filtering error-free data from being recorded into buffers. It measures the error rate to determine the suspect clock cycles when errors can be present, then it only captures the signals into buffers during those suspect clock cycles. Thus it expands the observation window through selective capture, leading to improved tracing efficiency. Another work [54] proposes *speculative debug insertion* to predict what signals will be useful during debugging for improved efficiency and observability. Unlike other flows where the instrumentation is added to help identify a bug that is known to exist, this prediction adds instrumentation from the first compilation, targeting fewer debugging iterations to improve verification efficiency. Therefore, this method trades resource overhead in the predicted instrumentations for reduced debugging iterations, leading to improved

overall productivity.

Some other works [43, 45, 97] also target using trace buffers to record the *in-situ* of HLS-generated designs, but provide different features to enhance the visibility of design. For example, Monson [43] proposes the use of Event Observability Port (EOP), which is essentially a top-level RTL port that corresponds to the output of a specific expression in the original source, to provide observability of source-level events from the corresponding HLS-generated hardware. It features selectively capturing the data only when an relevant event occurs, thus leading to more efficient memory usage for the tracing. Based on this idea, Monson [45, 97] proposes a source-to-source transformation method to increase visibility, in both simulation-based and execution-based scenarios. The transformation [45] features adding extra variables, such as a pointer parameter of a function and a global variable, together with Vivado HLS pragmas create the top-level ports representing the EOPs. This method shows that the visibility can be added to simulation through source-level transformations without modifying the timing, latency, or throughput. The extended work [97] is presented to justify that the area and clock period overhead incurred [45] is tolerable. It conducts a thorough experiment to quantitatively prove that with $25 \sim 50\%$ area overhead and $\sim 5\%$ increase in clock period, the instrumentation is feasible.

From these works based on using trace buffers to detect execution-based discrepancies we can see, these methods assist users in selection of signals to trace and associated instrumentation and/or source-level transformations to create debugging ports [44, 46, 53, 54, 43, 45, 97, 55], but generally concentrate on increasing the efficiency of signal tracing rather than assisting in diagnosing the functional mismatch.

SW-Like Debugging

In addition to the debugging methods based on HW/SW execution discrepancy, verification efforts in HLS have also worked on developing software-like debug environments. The feature set of the software-like debugger includes: single-stepping, breakpointing, setting/watching values of variables, and location of execution points. For example, Hemmert [51] proposes a GUI-based hardware debugger to support these

software-like debugging features. It also supports both the source-level and circuit-level views during the hardware execution. There are similar source-level debugging tools for the LegUp academic HLS tool [50, 44, 46]. Calagar [50] proposes *Inspect*, a GUI debugger on top of GDB to support software-like debugging features (like stepping and breakpointing). Different from Hemmert's debugger [51], *Inspect* [50] features automated HW/SW discrepancy detection, wherein *Inspect* communicates with concurrently running GDB and ModelSim processes to identify the first point at which a software execution of the program mismatches with the hardware execution, with correlation to C sources. Both software-like debugging features and automated detection of discrepancies significantly increase the verification efficiency, thus leading to considerable productivity improvement.

HW Assertions

An alternate approach has also synthesized C assertions into FPGA circuits during HLS for debug assistance [52, 98]. Although inserting user-predicted C assertions can be useful, we argue that an automated technique with visibility to all datapath operations is more flexible and superior for automating bug localization without user interaction.

Despite the productivity improvements from these works, there are still many remaining challenges in source-level debugging. In order to assist the source-level debugging of HLS-generated RTL efficiently for improving productivity, not only should execution mismatches be detected automatically, but they should also be localized quickly and precisely with as low overhead as possible.

2.2.2 Verification of HLS Tools

In addition to works that directly influence productivity, verification of HLS tools also indirectly improves productivity by supporting verification of advanced optimizations that improve tool QoR. HLS QoR has a significant impact on effort devoted to design iterations spent modifying behavioral specifications. Therefore, functional verification of HLS optimizations becomes critical to QoR and thus critical to overall HLS

productivity.

HLS tool verification research efforts explore both formal verification methods that attempt to prove formally that optimizations are always functionally equivalent. In contrast, simulation-based methods use a suite of benchmarks and data testcases and simulation-based verification that all benchmarks are processed correctly. We will first discuss research in formal methods before discussing simulation based tool verification.

The formal methods statically analyze the behavior of models described in both HLL and RTL descriptions, and perform equivalence verification by using techniques like sequential equivalence checking (SEC) [61] and bounded model checking (BMC) [64]. However, because formal verification usually relies on solving satisfiability (SAT) using a SAT solver, such as Chaff [65], the processing time increases exponentially with the design size, resulting in impracticality when designs become large and complex. Optimizations in HLS present a significant challenge for formal verification. As shown in Kim [99], the optimization of global code motions (GCM) considerably increases the difficulty of formally verifying functional equivalence between an HLL specification and the CDFG generated by HLS scheduling. Modern HLS tools include a variety of additional optimizations (Section 2.3), all of which increase the difficulty of equivalence checking.

Verification of HLS tools is similar in concept to verification of general purpose compilers, which is a known and challenging problem. Instead of verifying that the HLS tool or compiler executes without error, we must also prove that the output RTL (or binary) is always functionally correct given input specifications. Typical compilation frameworks such as LLVM [26] are not formally verified; projects attempt to verify some LLVM passes, but have not successfully produced a fully-verified compiler [66]. Projects such as CompCert [67] build compilation frameworks in Coq [68], a proof assistant language, so that the resulting compilation framework can be formally verified.

HLS tools are built on general compiler infrastructure such as LLVM. Formally verified compilers often lag behind general purpose compiler infrastructures such as

LLVM and GCC in terms of quality due to the challenge of formally verifying complex optimizations. Some prior work has attempted to formally verify functional equivalence between input high level source and output RTL [61, 62, 63]. However, complex CDFG transformations and hardware-oriented optimizations prove difficult to formally verify: many valid, important performance optimizations are rejected because the formal proof cannot verify functional equivalence. As the complexity of HLS optimizations increases, this problem will only be exacerbated, especially in a framework that has no underlying formal framework.

Compared to formal methods, simulation-based methods verify the functional correctness via simulation. Since the simulation uses real test vectors to verify the functionality, the complex transformations and optimizations which are rejected by formal methods can be accepted in the test flow, but there is significant dependence on benchmark selection and test vector selection to ensure that the simulation-based tests cover enough cases to have high confidence that the tool is bug free. Traditional simulation-based approaches may only find behavior discrepancies at the end of the simulation, millions of cycles after the bug is activated [61]. Long bug detection latency not only increases verification processing time, but also imposes huge human effort to correctly select/monitor a potentially large set of relevant signals and backtrace them for millions of cycles. As the capacity of SoC and complexity of design grows, this traditional simulation-based method becomes even more impractical for verifying large designs. Therefore, simulation-based methods must deliver both useful diagnostic information to facilitate the debugging process and reasonably reduce simulation time to speedup verification.

In addition to these two main categories of verification methods for HLS tools, there has been some complementary work that facilitates the test flow. Liu [69] proposes an automated test case generation technique for verifying/debugging HLS tools that generates random programs with user-specified features/characteristics to verify HLS tool correctness. Although automated test case generation helps verification, it does not aid in finding the bug when a functional mismatch is detected.

Other hardware verification approaches target post-silicon verification effort with

information about bug sources which might be related to HLS tools. The hybrid quick error detection (HQED) technique [70] extends the quick error detection (QED) [100] method for post-silicon validation of CPU-based systems to also validate HLS-produced cores by inserting detailed signatures that are used to validate correct synthesis of a high-level source. Although HQED can detect the existence of bugs at both the source-level and in HLS tools, it does not produce diagnostic information to assist in debugging.

2.3 QoR Optimizations for Productivity

In addition to works that directly address productivity via improvements to reuse and verification, there are many works that target optimizations for QoR improvement but have indirect impact on overall productivity. These works are proposed to optimize specific performance bottlenecks in the HLS flow, but also improve productivity through improved QoR. Therefore, in this section, we present a review of these works in HLL compilation, optimizations in scheduling, allocation and binding, and miscellaneous optimizations that address multiple portions of the HLS process.

2.3.1 HLL Compilation

As introduced in the HLS design flow in Figure 1.4, the HLS design flow begins with the same steps for software compilation of high level languages, the front-end and the compilation optimizations. These software compilation steps are responsible for transforming the statements in HLLs into IR and applying optimizations in passes [101], such as dead-code elimination, constant propagation, strength reduction, loop unrolling and loop rotation. For example, Xilinx's Vivado HLS (formerly AutoPilot [71]) and LegUp from the University of Toronto [83] are implemented within LLVM, and GAUT [102] is based on GCC [87]. The optimized IR serves as input to the following HLS stages and is thus critical to HLS QoR, and important to time and effort spent in design iterations for refinement.

However, optimization passes are created to optimize run-time of the software

not for optimizing hardware descriptions targeting architecture. This mismatch results in suboptimal HLS-generated hardware; there are many (56) transformation passes [101], and it is challenging to select a specific set of passes that meet hardware design objectives.

A prior work [103] presents an optimization using speculative global code motions (GCM) to improving the QoR of HLS. This work is based on their HLS framework, SPARK [104], and involves transformations such as re-ordering, speculation, and increasing the number of operations in HLL descriptions. It targets minimization of control flow latency in the HLL program, and features heuristic-guided GCM to reduce the number of finite state machine (FSM) states by 30% and latency in critical path cycles by 50%. Kurra [105] proposes a study of the effect of another compiler optimization, loop unrolling, on HLS QoR, and presents an exploration strategy to determine promising loop unroll values.

Based on the speculative GCM method [103], Gupta [106] explores an extended set of optimization transformations, including common sub-expression elimination (CSE), copy propagation, dead code elimination and loop-invariant code motion. They present a technique called *Dynamic CSE* to dynamically coordinate CSE and code motions and speculation during scheduling, so that the HLS-generated design can be parallelized. This work considers a bigger coverage of compiler optimizations that has impact on HLS QoR, but the re-ordering of operations is still based on the objective of minimization of the effects of the control flow [103]. Cong [107] proposes a study of the impact of compiler optimization phase-ordering on design space exploration and the quality of the generated RTL. It presents studies on three important optimizations: loop unrolling, software pipelining and array partitioning, and the effects of phase-ordering of multiple IR-level optimizations, such as dead code elimination, bitwidth minimization, constant propagation, and global common sub-expression elimination and so on. Based on the studies, it presents a simple heuristic, which iterates through a selected set of unroll factors and measures the drop in latency and the increase in area until the slope of area increase is greater than the slope of latency decrease, to quickly eliminate bad choices. Results show that it achieves 50% reduction in latency compared to the default settings in the

HLS tool, xPilot [32]. This work considerably extends the coverage of analysis on the impact of compiler optimizations on the HLS QoR, and starts using iterative heuristics to solve the phase-ordering problem. Huang [108] presents a study of the impact of a wider range of compiler optimizations onto HLS QoR. It explores: 1) the impact of each isolated LLVM pass, 2) the interdependency between passes, and 3) the impact of ordering. Meanwhile, an HLS-directed approach is proposed, where profiling is performed for each pass and transformations are accordingly accepted and undone when passes bring positive and negative impact to QoR, respectively. This work outperforms *-O3* standard optimization option in the metric of speed performance. Meanwhile, the automation of organization process of passes helps to reduce the design effort in selecting and re-ordering the suitable set of passes for the targeted performance objectives, thus improving the productivity.

2.3.2 Scheduling

As discussed in 1, scheduling plays an important role in the control flow generated by HLS and is critical to the QoR. Based on the characteristics of the applications to be scheduled, scheduling algorithms can be categorized into two groups: data-flow-based (DF-based) scheduling and control-flow-based (CF-based) scheduling [109].

According to the objectives, DF-based schedulings can be classified further into: 1) timing-constrained schedulings for minimizing resource cost; 2) resource-constrained schedulings for minimizing the number of control steps. Among timing-constrained schedulings, Force-directed scheduling (FDS) [110] targets a heuristic for resource optimization, such as reduction of functional units, registers, and buses by balancing the concurrency of the operations assigned to them. Sinha [111] extends force-directed scheduling by supporting parallel constructs and timed constructs to improve resource sharing in the HLS-generated hardware. For resource-constrained schedulings, list scheduling [112] and its extended works [113, 114] schedule ready operations sorted in a list into control states with available resources according to the priority function.

Path-based scheduling [115] proposes an as-fast-as-possible (AFAP) algorithm to address control constructs like conditional branches and loops to minimize the number

of control states. Bhattacharya [116] proposes a loop-directed scheduling (LDS) algorithm that produces schedules using a depth-first search (DFS), such that the expected number of clock cycles is minimized for execution of nested conditionals and loops. Lakshminarayana [117] proposes "Wavesched" scheduling targeted at minimizing the average execution time of control-flow intensive designs. It applies optimizations for parallelism within and across loop boundaries to a CDFG input.

There are works that target a versatile solution for both DF-based and CF-based scenarios and the support of various constraints. Cong [109] proposes a scheduler that supports a rich set of constraints, such as resource constraints, dependency constraints, frequency constraints, latency constraints, and relative timing constraints, by converting them into a system of difference constraints (SDC). It uses a unified mathematical programming framework to perform a variety of optimizations for objects like longest path latency, expected overall latency and the slack distribution. Thus the SDC-based scheduler is applicable to a broad spectrum of applications since it can generalize the as soon as possible (ASAP)/as late as possible (ALAP) and list scheduling techniques to schedule data-flow-intensive designs and perform global optimizations over complex control flows with low computational complexity at the same time. By efficiently supporting broadened application domains, where both data-flow-intensive computation and control-flow-intensive controller are included, it generalizes HLS to a much wider application range so that the development process of more applications can take advantage of HLS to speed up the design cycles, leading to significant productivity improvement.

Based on SDC-based scheduling, two extended scheduling methods [118, 119] are proposed to further optimize the resulting RTL design by using soft constraints and multi-cycle path analysis, respectively. Cong [118] defines soft constraints as the constraints which are not essential for the correctness of the functionality of the design, e.g. a constraint of cycle time which is slightly violated and still able to be met due to inaccuracy in estimation of interconnect delay and potential timing optimizations in following stages. Then a scheduler, which distinguishes soft constraints from hard constraints when exploring the design space, is proposed and applied to applications with a low-power scheduling problem.

Another work [119] proposes to analyze multi-cycle paths (MCPs), which are register-to-register paths connected by a combinational block with delay greater than one clock cycle. It generates a state transition graph (STG) to examine all reachable states. According to this STG, multi-cycle path analysis is performed, including use-define chain identification, available cycles calculation and time complexity analysis, to justifiably identify all reachable circuit states and generate multi-cycle constraints from the available cycles calculation for logic synthesis tools like Quartus II [33]. Both of these works achieve significant QoR optimizations, such as power saving [118] and improvement of execution time and resource utilization [119].

Therefore, these QoR improvement obtained by the optimizations in scheduling demonstrates that the design effort for a particular objective can be reduced if the optimization can efficiently improve the performance in that metric, and that the productivity can be improved because of the improved QoR meeting the constraints without further refinement process.

2.3.3 Allocation and Binding

Research in allocation and binding also demonstrates non-trivial QoR improvement which indirectly affects productivity, and is examined respectively.

Rim [120] presents an integer linear program (ILP) formulation for the allocation and binding problem to minimize wiring and multiplexer area. This formulation handles chaining, multi-cycle operations, pipelined modules, and conditional branches. Beidas [121] addresses the optimization of inter-procedural register allocation in HLS. By using the propagation of coloring constraints only in locally available colors, instead of directly using the massive global conflict graph, it achieves significant speedup compared to global solutions, and proves to be scalable to much larger applications. Brisk [122] extends Beidas [121] by proving the inter-procedural register allocation of HLS has an optimal, polynomial-time solution, and achieving that solution through an augmented static single assignment (SSA) form with launch and landing pads, which are duplicate operations placed before and after each procedure call.

Mujumdar [123] formulates binding into a network optimization problem and

proposes a heuristic solution that targets area reduction. By considering the floorplanning during the process, reduced interconnect area is achieved. Tomiyama [124] presents an evaluation model to predict the probability that the synthesized circuits will work at a specified clock period, and estimate the bound on the critical path. Based on this information, it proposes a statistical performance-driven binding, which maps the operations to functional units in such a way that the performance probability of the designed datapath is maximum, to achieve the reduced clock period. Chen [125] targets multiplexer reduction by using a k-cofamily-based algorithm to optimize register binding and port assignment. Cong [126] proposes pattern-based behavior synthesis to extract and utilize the regularity in applications for FPGA resource reduction. It defines a mismatch-tolerant metric, graph edit distance, as the minimum number of vertex/edge insertion, deletion, substitution operations to transform one graph into the other. Based on this definition, the pattern recognition algorithm exploits advanced subgraph enumeration and pattern pruning techniques to recognize patterns as labeled directed acyclic graphs (DAGs). Then, pattern-based binding guarantees that the corresponding nodes of pattern instances are assigned to the same function unit, leading to reduced allocation of resources. This work achieves a noticeable improvement in resource reduction with a reasonable latency overhead. It demonstrates that the effort for tweaking the trade-off between performance and area overhead can be efficiently saved by enabling/disabling the optimization, leading to productivity improvement. The extended work [127] achieves further resource reduction by taking into account the control-flows for pattern extraction, which leads to more convenience and efficiency in the trade-off process, thus more productivity, by allowing wider space for the automated trade-off exploration.

2.3.4 Miscellaneous

In addition to the optimizations related to a particular HLS process discussed above, there are other kinds of optimizations which address a specific bottleneck.

Cong [128] proposed an automatic memory partitioning technique to address the bottleneck of limited memory bandwidth, so that throughput and power consumption

can be improved. The partitioning algorithm generates partition candidates which balance array accesses to different memory banks. Then, throughput optimization is performed by going through a branch and bound search process to minimize the concurrent array accesses on each memory bank until the constraint of memory port limitation is met. In addition, power optimization algorithm reduces the dynamic power consumption by further partitioning the memory bank and grouping frequently visited elements into small memory banks. By using a similar approach [129], an iterative search step is adopted to find the partitioning solutions for the power optimization problem. This work achieves a significant improvement in throughput and power reduction with a moderate area overhead. It demonstrates that the automated iterations of the optimization can bring in significantly productivity improvement by replacing the manual effort in the refinement iterations for a particular bottleneck.

Zuo [130] proposes polyhedral model-based loop transformations to increase the opportunity for HLS [32] to perform intra- and inter-block optimizations. Risset [131] uses a polyhedral representation for nested loops to perform C-to-C preprocessing before synthesis to collect compiler advances for HLS optimizations, although it targets SPARK HLS framework [104] instead.

Wang [132] proposes an iterative improvement strategy to refine the cost function, which can be power, energy or area. Starting from an initial architecture, it generates modifications, such as clock selection, module and register binding, and runs through HLS to estimate the cost function using SCALP [133]. With the estimates calculated, the combination of modifications with improved cost is generated. Zheng [134] addresses the issue of sub-optimal RTL generation due to the inaccuracy caused by the fact that the actual data-path delay is often quite different from the estimated delay collected from benchmark profiling. It introduces a new flow to integrate with the logic synthesis tool and fast placement and routing (PAR) to obtain realistic post-PAR delay estimates for further improved maximal frequency. This work achieves noticeable improvement in both *Fmax* and execution latency. It also demonstrates that the improvement brought in by the optimization can efficiently reduce the negative effect caused by the inaccuracy in profiling, leading to reduced effort in collecting accurate timing characteristics for design refinement and overall

improved productivity.

There are optimizations for improving QoR in special application scenarios as well. Wang [135] targets the bottleneck of HLS tools that the multi-process system is handled with degraded QoR of HLS. It analyzes the criticality of each operation in the multi-process design and automatically partition the system-level resource constraints into local constraints for each pass. Mukherjee [136] proposes temperature-aware allocation and binding to minimize the maximum temperature that can be reached by an HLS-generated design, so that the design effort required for reaching reliability closure is reduced. These works proposed for improving HLS QoR in various scenarios enable productivity boost with wider adoption of HLS-based design methodology.

2.4 Conclusions

From the related works, we have discussed how overall productivity can be influenced by many factors in HLS-based design flows. The major factors, which directly and significantly contribute to productivity improvement are reusability involving IP integration, and verifiability involving both the source-level debugging of HLS-produced designs and the verification of HLS tools.

For reusability, the dominant methodology for IP integration is still conducted in the system level, where the integration process still requires substantial manual design effort, such as partitioning the HLL application into pieces that use and do not use IPs, exploration of various strategies for instantiation and interconnection. Although there are works that accept the behavioral level for IP integration, they either have a limited subset of IPs supported which are bound to unchangeable mapping to a proprietary library, or relies on a processor for compilation and execution rather than instantiation of IPs into HLS-generated circuits, or induces extra time and area overhead from the translation between RTL of IPs to C++ specifications. There is a need for a generic solution for IP integration in behavioral level, the same abstraction level of HLS, to efficiently integrate as many types as possible user-specified IPs to take the advantage of reusability from IPs for productivity improvement.

For verifiability, because of the challenges in formally verifying the HLS-produced

designs in formal methods, and the difficulties in interpreting non-human-readable RTL and back-tracing from the detection of a discrepancy to the activation source of the bug in the simulation, productivity is considerably limited due to time-consuming verification. There is a need to overcome these challenges and provide functional verification support to accelerate the verification and debug process. On the other hand, given the implicit yet non-trivial impact on productivity by HLS optimizations, the challenges in formally verifying HLS tools built on non-formally-verified compiler infrastructures, and the huge effort devoted to bug tracing, it is also necessary to provide a solution with both efficient detection of bugs in reduced time to speedup the verification, and useful diagnostic information to facilitate the debugging.

In addition to these main factors, the indirect impacts on productivity caused by the optimizations for QoR improvement have also been examined. They either target improvement of QoR from different perspectives of HLS methodology (e.g. iteration flow or pre-HLS transformation), or specifically address a bottleneck issue for improvement in a particular metric, or adapt the HLS optimizations to various scenarios. As a result, the QoR improvement achieved indirectly impacts the effort and time devoted in the design iterations and thus enhance the overall productivity in some degree, although it is not intentionally targeted at the improvement of the productivity directly.

In this thesis, we focus on addressing challenges and remaining productivity bottlenecks due to reusability and verifiability in HLS tool flows.

Chapter 3

Background

In this chapter, we introduce the VAST HLS framework [137], on which we will build the IP integration and verification features that will be presented in the later chapters. We will firstly describe the overall flow of the VAST framework in its entirety and the important components that are relevant to the discussion presented for productivity improvement in later chapters. Based on this background, we will introduce the plans of modification in the following corresponding chapters. The behavioral-level IP integration, presented in Chapter 4, extends the VAST HLS framework, with features such as user-specifiable IP mapping, IP handling in various levels of the framework. The AutoSLIDE cross-layer verification framework in Chapter 5 implements coarse-grained and fine-grained instrumentation and backtracing based on VAST for source-level debugging. The trace-based verification in Chapter 6 also extends the VAST framework by introducing automated insertion of verification code to facilitate the debugging of HLS tools.

VAST is based on the LLVM compiler infrastructure, similar to LegUp [138]. VAST accepts C/C++ source code, platform-specific constraints including area and latency of elementary instructions, and user timing constraints as input and produces RTL in Verilog, an RTL testbench for the design, and logic synthesis constraints. The VAST framework supports both Quartus II and Vivado logic synthesis backend tools.

3.1 VAST HLS Flow

We first present the sequence of steps in the HLS flow (Figure 3.1). In the next section, we will discuss the internal representation, which will be important for later discussion of IP integration in Chapter 4.

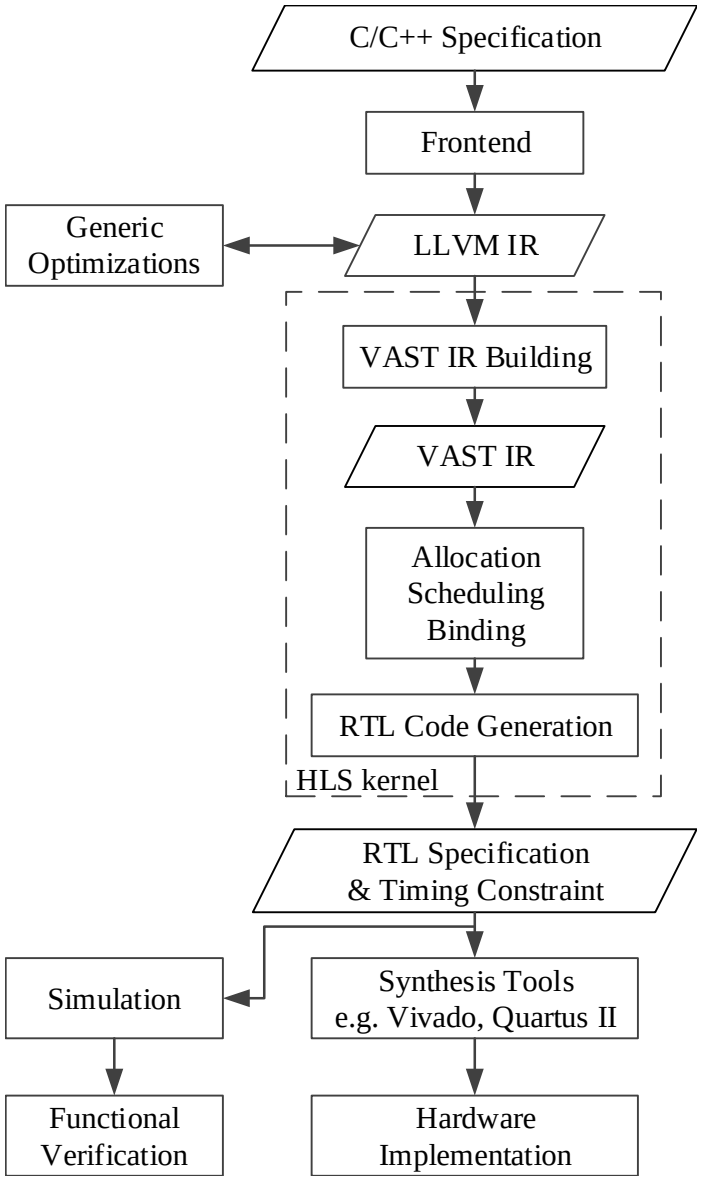

Figure 3.1: VAST HLS Flow

3.1.1 Parsing and HLS-independent optimization

The framework uses Clang [139] as the front-end to parse the source code and perform HLS-independent compiler optimizations. We use C/C++ descriptions as the source code input to Clang, which is a production-quality C++ compiler that is able to produce highly optimized specifications in LLVM-IR for later HLS-related transformations.

3.1.2 VAST IR Building

From the LLVM IR produced by Clang, the framework builds VAST IR which represents hardware-level information to support optimizations such as bit-level manipulation [140]. During this step, we modify IR building to include user-specified IP mappings, and characterization of IP area and latency. We will discuss IP specification and integration in more detail in section 4.1.

3.1.3 Allocation

The LLVM IR, in static single assignment (SSA) form, is used to allocate necessary resources. Resources such as external memories, I/O and external IPs must be serialized to guarantee correctness; thus, we perform binding before scheduling to ensure correct serialization.

For memory allocation, unified memory spaces are separated into multiple disjoint sub-spaces using LLVM's pointer alias analysis [141] so that each sub-space can be allocated an independent memory bank, which improves both area (due to reduced multiplexor/sharing hardware) and latency (due to increased parallelism) at the cost of increased memory banks.

3.1.4 Scheduling

The framework uses a modified version of the system of difference constraints (SDC)[142] scheduler that supports inter-basic block parallelism [143], multi-cycle analysis and chaining [137, 144], and global code motion [143].

3.1.5 Binding

VAST uses structural hashing-based binding algorithm to eliminate redundant structures in the design. For IPs with variable-latency or external interfaces, a single IP instantiation will be shared by all instructions/functions using that IP, as discussed in Chapter 4.

3.1.6 RTL Code Generation

VAST HLS generates RTL in Verilog together with an RTL testbench, and scripts for simulation and synthesis. In Chapter 4, we modify code-generation to also dump information about the IPs used to assist in generation of the RTL testbench and automated instantiation of external IPs. For Altera and Xilinx IPs, we also use this information to automate running Altera or Xilinx's IP-core generation tools. Because of the previous analysis and processing of the user-specified IPs, these automated steps considerably reduce the manual effort in the creation of testbenches, selection of IPs and the execution of IP generation tools, thus leading to further improved productivity. In Chapter 5 and Chapter 6, we also modify code-generation to generate debugging information for the verification flows.

3.2 The VAST Intermediate Representation

The VAST IR supports compilation optimizations such as dead-code elimination, as well as timing estimation, scheduling, and binding. To effectively partition optimization and analysis of instructions, the VAST IR consists of four layers that **explicitly** represent the design at the behavior, control-flow, data-flow, and hardware architecture levels. Each layer represents different types of information and supports different types of optimization passes.

An example of the IR is given in Figure 3.2. The behavior level contains instructions from LLVM IR, data-flow level contains LLVM computation instructions, control-flow contains LLVM non-computational instructions, and hardware architecture contains mappings to register, functional unit, memory and I/O shared resources.

These layers provide detailed information from different perspectives for the framework to apply various analyses and optimizations to each of the steps in VAST HLS flow.

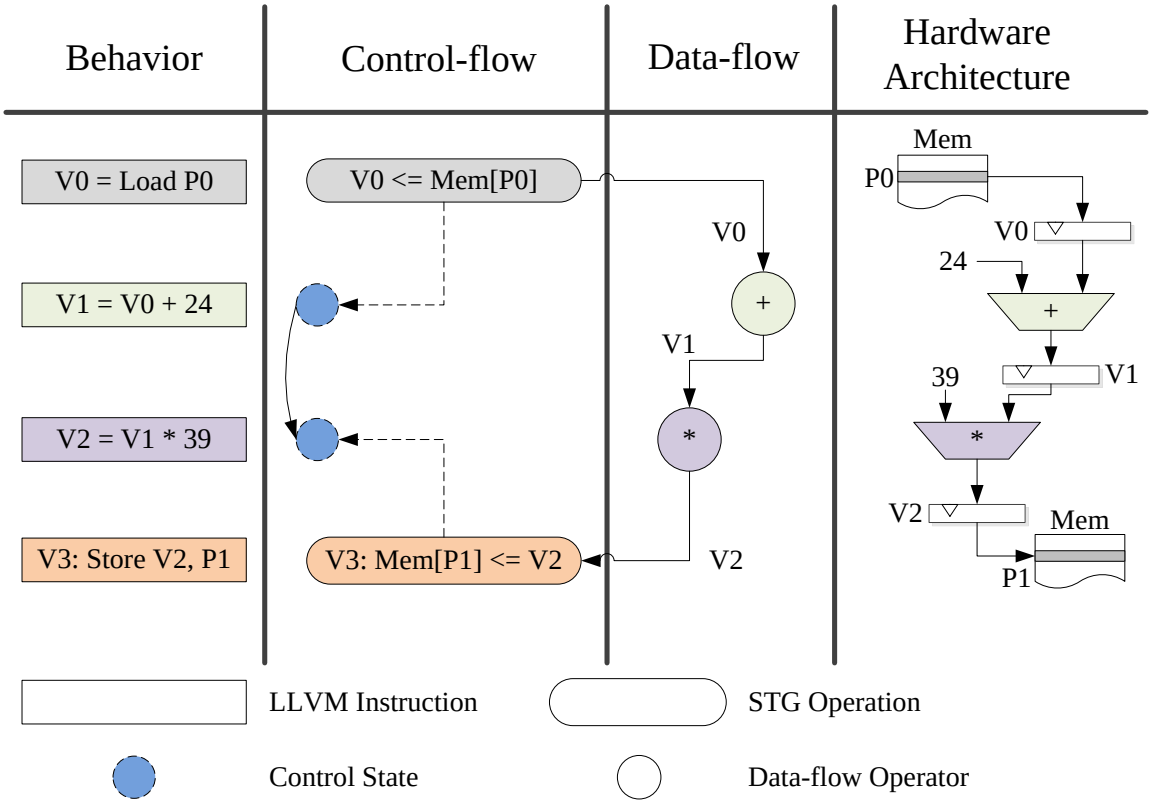

Figure 3.2: VAST IR

3.2.1 Behavior Level

The behavior level is inherited from LLVM IR; through analysis at the behavior level, we use high-level information such as pointer aliasing, the range of induction variables in loops, and control-flow dominator tree. This information is used during scheduling.

3.2.2 Control-flow Level

A state transition graph (STG) represents control state transitions. A control state in the STG contains *instructions* (nodes in the "Control-flow" column of Figure 3.2) which contain register assignments. A register assignment can define values in the data-flow (e.g. V0 in Figure 3.2). LLVM instructions in the behavior level are associated with instructions in the STG. Using behavior-level information, transformations on the control-flow level generate high-quality schedules (e.g. using pointer aliasing analysis to reorder memory accesses).

3.2.3 Data-flow Level

Directed-acyclic graphs (DAGs) represent computations and data-dependencies, where each node in the DAGs is an *operator*. The data-flow level contains a bit-mask for each instruction to specify compile-time known bits in each instruction, and nodes are in SSA form [145] to allow precise propagation of known values across instructions and allow global analysis of data dependence.

3.2.4 Hardware Architecture Level

The hardware architecture level represents physical resources in the hardware architecture such as registers, functional units, memories and I/O ports (e.g. V0, V1 and V2 as registers, adder and multiplier as functional units, and Mem[P0] and Mem[P1] as memories in Figure 3.2). In the initial SSA-form, every instruction has its own registers and functional units, and during binding these units may be merged to share resources in the hardware architecture.

3.3 Conclusions

We introduced our VAST HLS framework in this chapter to provide the background for our later discussion of the solutions to the critical challenges, i.e. behavioral-level IP integration, source-level debugging of HLS-produced design and the verification

of HLS tool. The overall flow and IR of VAST are described in detail so that we can directly discuss the specific updates or modifications on them for each solution in following chapters.

Chapter 4

Behavioral-Level IP Integration

This chapter addresses the challenges of IP integration in HLS-based design flows, and discusses our solution, behavioral-level IP integration, for productivity improvement. Prior works of IP integration instantiate and connect RTL IPs with other system-level components after the HLS process. Despite the promised productivity and performance gains, IP integration in HLS-based design flows expose multiple challenges.

First, the allocation, scheduling and binding steps play a decisive role in the exploitation of the parallelism of HLS-generated RTL implementations. In post-HLS (system-level) integration, these HLS steps can only take into account the portions without IPs in an application for parallelism exploration. Because system-level integration happens after the HLS process, the necessary information of IP characteristics, such as latencies and operations-to-IPs mapping, for the scheduling of IP-implemented operations is not yet available, and the HLS kernel loses the opportunity to achieve better parallelism by scheduling other operations in parallel with IP operations wherever the data dependence allows. Thus, the HLS kernel can only make conservative assumptions (e.g. the latency of IP is unknown and all IP-implemented operations are mapped to the same IP instantiation) to generate the scheduling results, thus leading to limited parallelism in the HLS-produced RTL implementation.

Binding is closely related to scheduling, and scheduling results with limited parallelism also have non-trivial impact on the binding. Serialized operations which are

scheduled according to conservative assumptions such as unknown IP latencies result in multiple operations sharing one physical IP instantiation. In addition, the number of IP instances determined by allocation directly impacts the binding. In system level integration, allocation is manually decided by designers, each IP is instantiated once and shared among callers, and a designer can explore multiple instantiations by creating duplicate function implementations (one per instantiation) so that callsites can be manually partitioned between the function instantiations. However, it is impractical for users to manually explore different numbers of instantiations and different permutations of binding callsites to instantiations. Therefore, it is impractical for system-level integration to address allocation and binding limitations on parallelism.

Next, in system-level integration, because non-synthesizable functions, such as C/C++ math library functions and I/O functions, are heavily used in software, the user must partition the application into portions that do and do not make use of non-synthesizable functions, so that the portions without those non-synthesizable functions can be accepted as synthesizable input to HLS processing. Then, portions with non-synthesizable functions need to be handled separately, by either removing non-synthesizable inputs or developing appropriate synthesizable implementations. However, this partitioning results in substantial effort in handling those non-synthesizable functions. First, users need to manually extract all non-synthesizable calls by either eliminating calls or creating interface ports to allow the synthesizable functions to communicate with non-synthesizable code for function calls. If the function calls are in a deep subroutine, creating and exposing the ports require changes in multiple levels of hierarchy in the call graph, and thus substantial effort. In addition, if it is not feasible to export the I/O ports for a non-synthesizable function, e.g. functions without input or return values, creating the extra ports for connection to system-level non-synthesizable IPs is not practical.

As discussed in Section 2.1.2, HLS tools prior to this work only integrate a small fixed set of IPs, but neither target generalized integration of third-party IPs, nor allow user management of function/instruction-to-IP mappings. Without a generalized, user-specifiable integration method, a wide range of reusable IPs cannot be utilized for further improved productivity.

For functions without HLS-appropriate implementations, such as mathematics, I/O and debug library functions which are are heavily used in software for functional verification, this lack of user management significantly limits the capability of HLS in taking advantage of these software features. As a result, the user must provide HLS-compatible C/C++ function implementations, or partition the application to separately handle them in system-level as discussed in the challenge of partitioning, or eliminate calls to those library functions in order to produce any hardware output, thus limiting the productivity by requiring these extra efforts.

Therefore, it is challenging for this post-HLS (system-level) integration strategy to effectively achieve IP integration with good QoR in parallelism, which affects productivity due to the need for many design iterations. In addition, the substantial additional effort required by partitioning reduces HLS-based design flow efficiency, and the non-generalized integration limits IP integration to a narrow set of IPs instead of the wide range of reusable IPs.

Therefore, we present our behavioral-level IP integration to take advantage of the analysis of HLS kernel to exploit parallelism and accelerate the process of reaching HLS-compatible C/C++ specifications without the need for partitioning. By reducing design effort, design cycles in space exploration and QoR refinement are accelerated, thus bringing significant productivity improvement.

In this chapter, based on our behavioral-level IP integration publication [94], we develop a generic IP integration framework for HLS that allows user-specified function/instruction-to-IP mapping to integrate existing RTL IPs. Existing RTL IPs can be fixed latency or variable latency, which is considered for both IP interfacing and operation scheduling. We also support software library functions by converting them into non-synthesizable IPs via automated translation or direct integration of C/C++ functions. Finally, IPs can be internally instantiated (behavioral-level integration) or externally (system-level integration) based on users' choice. Using this framework, we present four case studies to demonstrate the flexibility, utility and productivity improvement of our IP integration solution. This framework contributes to HLS and behavioral-level IP integration with:

- The first generalized IP integration supporting fixed- and variable-latency IPs,

and both synthesizable and non-synthesizable IPs.

- A flexible intermediate representation that tracks data-, control- and resource-dependencies for IPs.
- User-specifiable function/instruction to IP mapping.
- User-specifiable internal (behavioral-level) or external (system-level) integration.
- Four case studies demonstrating IP-integration-enabled development flows.

With our generalized IP integration, we extend development productivity support to allow a wide range of development flows including incremental HLS, instantiation and comparison of multiple IP-vendor implementations, use of non-synthesizable IPs for debugging, verification, and system-level simulation, integration and comparison of fixed- and variable-latency implementation options, user-specifiable internal or external IP integration and hierarchical application of HLS in a large application. The quick and automated exploration of various IP implementations, the easy exploration of various constraints (e.g. the number of instances), and the useful support of non-synthesizable functions for early verification all lead to considerable savings in design effort, thus achieving significant productivity improvement.

The rest of this chapter is organized as follows. Section 4.1 discusses our methodology to integrate IPs into our HLS flow. Section 4.2 presents the case studies to demonstrate the improvement in productivity and characteristic benefits of this method.

4.1 IP Integration in VAST HLS

We build our IP integration based on VAST HLS, which is discussed in Chapter 3. We first discuss the goals and principles for good IP integration that will guide the design and implementation of IP integration. These goals help to overcome the challenges discussed above, such as parallelism, partitioning as well as generalization for IP integration, so that further improved productivity can be achieved. Ideally, IP integration for HLS should:

- Support both fixed- and variable-latency IPs
- Be extensible to arbitrary functions or instructions in high-level source
- Support IP integration without limiting or affecting other HLS optimizations
- Support internally instantiated and external IP interfaces
- Support synthesizable and non-synthesizable IPs

Cumulatively, these goals describe an IP integration technique that allows a wide variety of third-party and user-generated IPs with high flexibility in the implementation and usage of these IPs while maintaining full compatibility with the entire set of HLS optimization features. The flexibility in the implementation and usage of these IPs guarantees that the generic IP integration method can be applied to any user-specified function/instruction-to-IP mapping to allow fast and automated exploration of various IP implementations, thus achieving significant productivity improvement. The compatibility with HLS optimizations guarantees that the integration of IPs does not prevent HLS from reaching optimized QoR, which is critical to the effort and time devoted into design iteration. Therefore, this compatibility introduces productivity improvement without imposing any overhead in other aspects related to productivity.

Our implementation of general IP integration has two main components: an extensible function/instruction to IP mapping to allow users to specify function/instruction candidates to be implemented by IPs and the corresponding IP implementations in VAST HLS, and extensions to our existing intermediate representation to represent functions or instructions implemented by IPs. Both of these two components are built into the VAST HLS flow discussed in Chapter 3. As shown in Figure 4.1, the newly added processes are highlighted in orange and the modified processes in VAST HLS are in blue.

4.1.1 Extensible Function/Instruction to IP Mapping

The first step in IP integration is a user-provided specification of functions and instructions that should be mapped to IPs. As shown in Table 4.1, this mapping

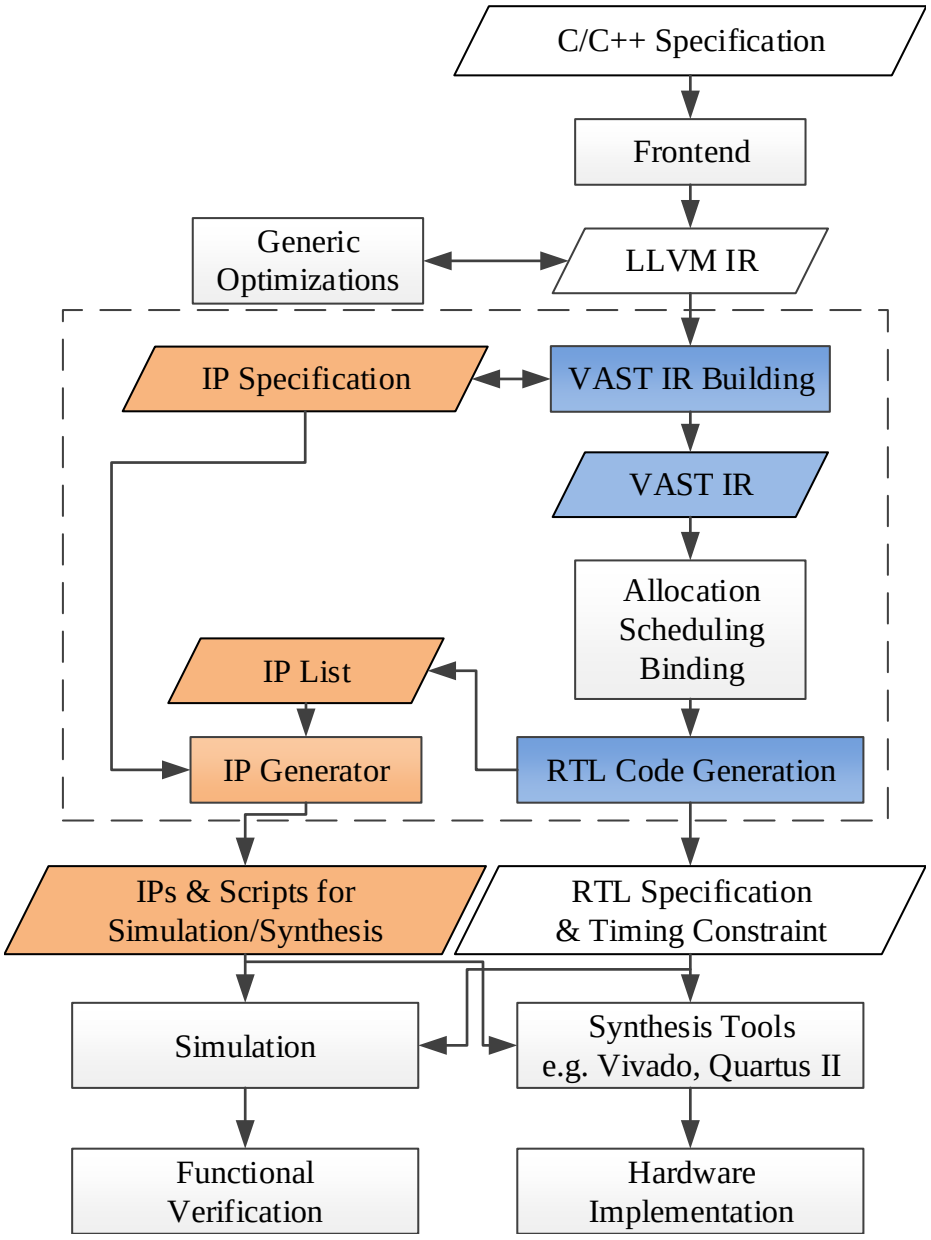

Figure 4.1: IP Integration in VAST HLS Flow

provides characterization information on latency, area, and a wrapper module netlist that conforms to our fixed- or variable-latency interface for internally instantiated IPs, i.e. the IPs instantiated within the HLS-generated top module for the application. Each function or instruction that will be mapped to an IP must provide the

Table 4.1: IP Specifications

Attributes	Descriptions
Name	Name of the instruction/function which will be implemented in IP.
Signature	Signature of the instruction/function including the return type, the number and types of operands/arguments.
Latency	Clock cycles required by the IP to finish a computation, when a negative latency specifies a variable-latency IP.
Area	Resource usage of the IP in terms of LUTs.
Externality	Flag for specifying whether an IP is external to the whole design.
Netlist	Wrapper module that instantiates the IP conforms to either the fixed- or variable- latency interface.

information as in Table 4.1. This specifies mapping between functions/instructions and IPs, whether an IP is fixed- or variable-latency, and internal (behavioral-level) or external (system-level) integration. Table 4.2 lists the interfaces for both fixed- and variable-latency IPs. As shown in Table 4.2a, fixed-latency IPs follow a simple clocked input-output interface; variable-latency IPs use start-finish handshaking signals (Table 4.2b). Internal IPs may use either fixed- or variable-latency interfaces; external interfaces always use the variable-latency handshaking interface.

The netlist can contain any instantiation that correctly uses the fixed- or variable-latency interface (Table 4.2). Following this interface, the contents of the wrapper can either simply instantiate the desired IP, build a controlling finite-state machine (FSM), or integrate more complex non-synthesizable code. In our case studies, we will show examples of synthesizable IPs using Altera's Floating Point IPs, non-synthesizable IPs (C/C++ library functions linked to a Verilog module through DPI), and hybrid modules (Xilinx FP modules, with encrypted IP cores used for synthesis and DPI-based functional simulation).

Although our two interfaces are simple interfaces for IP integration, they support a wide range of IPs: many IPs can be designed or used as fixed (worst-case) latency IPs, and IPs that use common bus interfaces such as AXI, Avalon, or Wishbone can often be instantiated in a wrapper that follows the handshaking protocol. For example, Xilinx IPs, which are compatible with AXI4 interface, have a 'tvalid' signal for every data input and data output channel. When these IPs are instantiated in

Table 4.2: IP Interfaces

Fixed Latency Interface	
Name	Description
clk	module clock – all IPs must be register inputs and outputs
rstN	module reset – IPs can optionally reset registers
inX	0 or more input ports, with width specified in instruction/function signature
outX	0 or 1 output port, with width specified in instruction/function signature

(a) Fixed Latency Interface

Variable Latency Interface	
Name	Description
clk	module clock
rstN	module reset
start	start computation – HLS-generated top module asserts to tell IP to register inputs and begin computation
fin	finished computation – HLS-generated top module waits for IP to assert fin, signifying outputs can be read
inX	0 or more input ports, with width specified in instruction/function signature
outX	0 or 1 output port, with width specified in instruction/function signature

(b) Variable Latency Interface

the wrappers for variable-latency IPs, the 'start' signal can serve to assert the 'tvalid' signals for the input channels to initiate the operation of IP, and the 'fin' signal can be asserted by the 'tvalid' signal of the output channel to denote completion of that operation.

4.1.2 Extensions of the VAST IR

In order to support third-party IPs, we make modifications to each of the levels of the intermediate representation.

At the behavior level, we use the function/instruction-to-IP mapping from the user script to detect functions/instructions that will be implemented by IP blocks, with either fixed or variable latency. Once a function/instruction is found in the

mapping, it will be tagged for proper handling in the other three levels.

At the control-flow level, IPs for fixed-latency instructions are treated identical to elementary instructions such as additions or multiplications. For each variable latency IP, there is a waiting state machine as shown in Figure 4.2. The core FSM asserts the IP's 'start' signal, and then waits to receive a return. After the IP completes computation, it asserts the 'fin' signal, which is received by the waiting FSM. Because the same IP may be used multiple times, there are disambiguation flags to denote the correct next state in the core FSM.

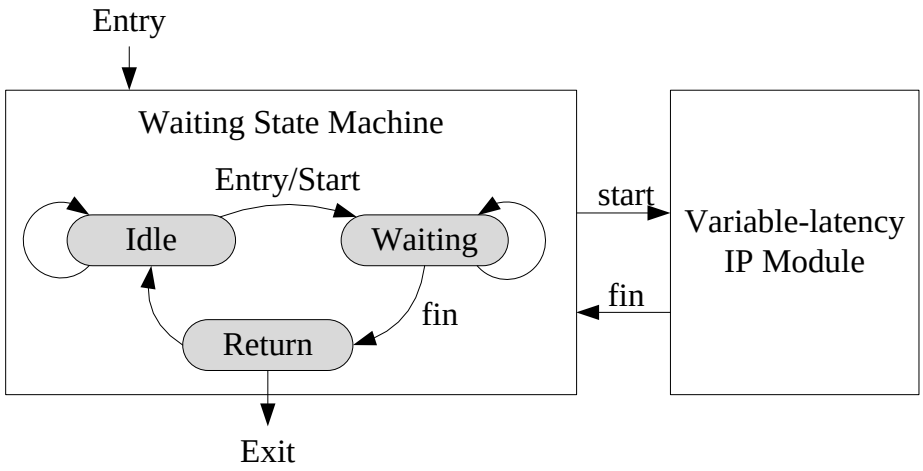

Figure 4.2: Waiting State Machine for Variable-Latency IPs

At the data-flow level, fixed-latency instructions are represented as computations with user-defined latency. Fixed-latency IPs are in SSA-form and can be instantiated for multiple times or optionally shared according to the scheduling result. Variable latency IPs may not be part of optimizations such as instruction-chaining or parallelization between multiple instantiations, where a fixed upper-bound on latency is required.

Finally, at the hardware architecture level, according to the latency information in user's IP specifications, fixed-latency IP instructions will become one or more instantiations of the IP-block, and variable-latency instructions will become a single instantiation of the IP-block shared by all users. The IP's inputs, outputs, and control signals are registered, and access to the input and output registers may be

shared through multiplexing.

4.1.3 Testbench Generation and Validation

In addition to generating instantiations in the Verilog output, our HLS engine generates a list of IPs used in the design to support automated testbench generation. In addition, we develop scripts that can automate use of Altera or Xilinx's IP generation tools to automatically create the desired IPs. To incorporate additional IPs, a user can either provide copies of the desired IP blocks and the script will copy them to the HLS-generated output directories, or provide appropriate commands to an external IP-generation tool.

For externally integrated IP cores, we use the list of IPs and interfaces to automatically generate a testbench that instantiates the HLS-generated design, external IP cores and the testbench. Because of the HLS being aware that which IPs need to be externally instantiated, the partitioning of external IPs, such as variable-latency IPs, non-synthesizable IPs, is automatically conducted, thus leading to considerable savings in manual effort for the partitioning process. In addition, testbenches are automatically generated and self-verifying if the behavioral-level application uses the return value of the top function to signify success or failure in execution, e.g. zero for success and non-zero for failure, thus leading to additional contribution to productivity improvement. Applications that use more complex features such as SystemVerilog DPI may require a specialized sequence of commands to compile and link code for simulation; thus, we also automatically produce simulation and synthesis scripts to reduce user effort in validating HLS results using ModelSim or Xilinx Vivado simulators, and Altera Quartus or Xilinx Vivado logic synthesis. The linking to SystemVerilog DPI not only provides much more software libraries (e.g. stdlib, math) that save the considerable effort in implementing the HLL functions in applications for functional simulation, but also speeds up the simulation process significantly by using the C/C++ model for much faster validation [146]. In addition to these benefits in productivity gains, the automated generation of simulation and synthesis scripts also contributes to productivity improvement by reducing the user effort in manually

going through the following simulation and synthesis processes.

4.2 Case Studies

In the previous sections, we introduced a generalized IP integration framework. In order to evaluate this IP integration framework, we set our evaluation metrics as the features that are critically important to the integration productivity, such as flexible, user-specifiable IP mappings, support for non-synthesizable functions, fixed-/variable-latency IPs, internal/external IPs, and user-specifiable target FPGA platforms. We will now perform several case studies to demonstrate how IP integration supports a variety of improved HLS-based design flows including exploring and comparing IP implementations, incremental and iterative HLS, hierarchical HLS, and use of non-synthesizable IPs to support debug and verification. As discussed in the motivations in Chapter 1 and the challenges in the beginning of this chapter, these features take advantage of much wider range of IPs, quick design space exploration with HLS-automated IP integration, and non-synthesizable library functions to overcome the productivity challenges and achieve productivity improvement.

4.2.1 Case Study 1: Floating-Point IPs

In our first case study, we demonstrate general IP integration as a means to support instructions and functions not part of the Verilog syntax. Although floating-point designs are common, they are constructed with IP blocks of the underlying integer instructions. To support floating-point instructions, we use the user-specified mapping to instantiate appropriate IP blocks for floating-point instructions and library functions.

We target both Altera and Xilinx platforms to demonstrate our integration framework can support user-specifiable target platforms. In order to evaluate our support for the critical the metrics presented earlier, complex benchmarks which include IP-supported instructions/functions and non-synthesizable functions are needed. We select seven benchmarks from the Rodinia benchmark suite [147], which represent

Table 4.3: Number of Floating Point Operations/Functions in LLVM IR

	The Number Of IP-supported Operations/Functions														
benchmarks	+	-	*	/	FP2Int	Int2FP	Comp	FPExt	FPTrunc	exp()	sqrt()	srand()	rand()	printf()	assert()
backprop_kernel	49	18	55	7	0	11	6	25	13	7	0	1	9	1	1
nn	1	4	2	0	0	0	2	3	1	0	1	0	0	0	0
hotspot	38	27	28	40	1	2	0	0	0	0	0	0	0	0	0
lavaMD	11	5	14	5	1	5	0	1	1	1	0	0	0	0	0
lud	1	4	3	1	0	0	2	2	0	0	0	0	0	0	0
srad	16	7	20	9	0	3	3	9	5	1	0	1	1	1	0
cfd	93	7	82	8	1	1	0	20	11	0	7	0	0	0	0

large, complex floating point applications that use both elementary operations and complex functions such as exponential, square root, and logarithm, as well as non-synthesizable library functions such as random number generator and assertion. Table 4.3 lists the applications and the number of instructions/functions of each type in the application.

Using the OpenMP version of each application [147], we modify each benchmark for HLS compatibility, and for introducing logic to ensure the application is self-verifying by using the return value of the top function with the automatically generated testbench.

For each application, we perform high-level synthesis, and automatically generate testbenches and simulation scripts to perform ModelSim and/or Xilinx Vivado simulations. We validate results with functional simulation and then perform logic synthesis. We target an Altera Stratix IV EP4SGX230K using Quartus II 13.0 and a Virtex 7 XC7V2000T using Vivado 2014.4. Due to Xilinx's encrypted IPs, we use Vivado simulator for simulations of Xilinx designs, and ModelSim 10.1d for simulations of Altera designs.

To demonstrate the compatibility with HLS optimizations, for each application, we synthesize using no additional HLS optimizations (No-Opt), and with multi-cycle path analysis [137], and global code motion [143] VAST HLS optimizations (HLS-Opts). With our current one-to-one binding algorithm, fixed-latency IP blocks are not shared, so there may be multiple instantiations running in parallel; variable-latency IPs and externally instantiated IPs are always shared.

Table 4.4: Results: Latency and Fmax

	Latency in cycles		Fmax in MHz	
benchmarks	No-Opts	HLS-Opts	No-Opts	HLS-Opts
Altera Platform - Stratix IV EP4SGX230K				
backprop_kernel	160088	157198	147.32	140.59
nn	787369	671886	218.25	138.77
hotspot	200	200	135.83	141.38
lavaMD	771089	741389	163.72	157.88
lud	3891360	3576951	146.31	147.86
srad	18506	17165	139.45	140.08
cfd	22950	23093	138.50	138.70
Xilinx Platform - Virtex 7 XC7V2000T				
backprop_kernel	245740	242849	257.776	182.205
nn	971530	738851	286,670	164.249
hotspot	420	420	296.619	281.426
lavaMD	1502974	1473293	292.113	194.995
lud	5197784	4872040	277.521	202.950
srad	33068	32429	318.032	324.009
cfd	35768	35673	215.827	287.329

Table 4.5: Results: Resource Usage

	Altera ALUTs		Altera Registers		Altera BRAM in bits		Altera DSP 18-bit	
benchmarks	No-Opts	HLS-Opts	No-Opts	HLS-Opts	No-Opts	HLS-Opts	No-Opts	HLS-Opts
backprop_kernel	34,497	34,235	33,773	33,235	157,248	157,248	328	328
nn	3,382	3,168	3,920	3,677	684,864	684,864	8	8
hotspot	46,907	46,925	42,252	42,252	728	728	616	616
lavaMD	38,191	39,853	38,114	32,280	88,064	88,064	166	166
lud	4,636	4,475	4,620	4,438	266,752	266,752	28	28
srad	25,570	25,542	25,010	24,655	42,164	42,164	306	306
cfd	92,390	92,405	107,956	107,956	68,768	68,768	780	780
	Xilinx Slice LUTs		Xilinx Registers		Xilinx RAMB36 in bits		Xilinx DSP48E1	
backprop_kernel	25,001	24,863	44,926	44,353	294,912	294,912	215	215
nn	2,086	1,984	3,915	3,678	1,271,808	1,271,808	10	10
hotspot	59,699	59,712	110,940	110,905	147,456	147,456	139	139
lavaMD	26,635	24,313	43,571	38,110	258,048	258,048	196	196
lud	3,192	3,092	5,621	5,429	294,912	294,912	16	16
srad	19,877	19,866	38,577	38,349	313,344	313,344	146	146
cfd	62,861	62,889	131,165	131,122	202,752	202,752	448	448

Performance and area data for all of the benchmarks on Altera and Xilinx platforms are shown in Table 4.4, 4.5. These results demonstrate that, with our easy user-specifiable function/instruction to IP mapping, users can easily choose various IP sources for implementing the same application without changing the input HLL

descriptions or manually handling the instantiation and connection of IPs. This allows the evaluation and comparison of IPs to be conducted in a much easier and faster way. In addition, the results also show that the IP integration framework is independent of the target platform, thus facilitating the comparison of the implementations with various IPs in different platforms. All these features lead to higher efficiency and increased productivity in the design space exploration with various available IPs and target platforms. In addition, these results also show that, our behavioral-level IP integration is compatible with various HLS optimizations, so that the integrated IPs can also be taken into account when various HLS optimizations are applied to improve the QoR. This compatibility takes advantage of both the reusability of IPs and HLS optimizations to quickly deliver optimized QoR for reduced design iterations, thus contributes to the overall productivity. These results already take into account the external IP integration, where the waiting state machine is used to interface in between the HLS-generated main module and the external IP instances. Therefore there is an overhead of two clock cycles in each invocation of external IP instances, one cycle for the 'start' signal and the other for the 'fin' signal. However, this minor overhead is less critical because it only accounts for a small percentage compared to the execution time of these IP instances, which can be dozens or hundreds of clock cycles. Although this case study demonstrates IP integration with floating-point blocks, any IP with fixed-latency can be simply integrated using this technique.

4.2.2 Case Study 2: Non-synthesizable Functions

In addition to integration of synthesizable IPs, it can be useful for development to integrate non-synthesizable IPs. Non-synthesizable IPs may be used for debug and validation of internal signals through the use of $\$display()$ and $\$assert()$ simulation-only functions, or CPU implementations of functions such as $srand()$ and $rand()$ through the SystemVerilog DPI interface. Although these non-synthesizable IPs will need to be replaced or removed before final synthesis, development flows may perform more rapid validation through use of the exact CPU implementation for functional verification, or detailed debugging and profiling.

In the tested set of Rodinia benchmarks, several use math library functions that do not have exact equivalent hardware implementations. *backprop_kernel* and *srad* use the *srand*() and *rand*() functions; although random number generator IPs are available, it is desirable to first verify the HLS-generated design using the CPU library implementation to guarantee correct operation with an identical random number sequence. For these benchmarks we use DPI to link the existing C/C++ stdlib library functions into the instantiated IP block as shown in Figure 4.3. In addition, some non-synthesizable IPs can also be supported directly by Verilog/VHDL tasks or functions without linking to DPI library functions, e.g. *printf*() in Figure 4.3 is directly supported by using $*display*() task in Verilog.

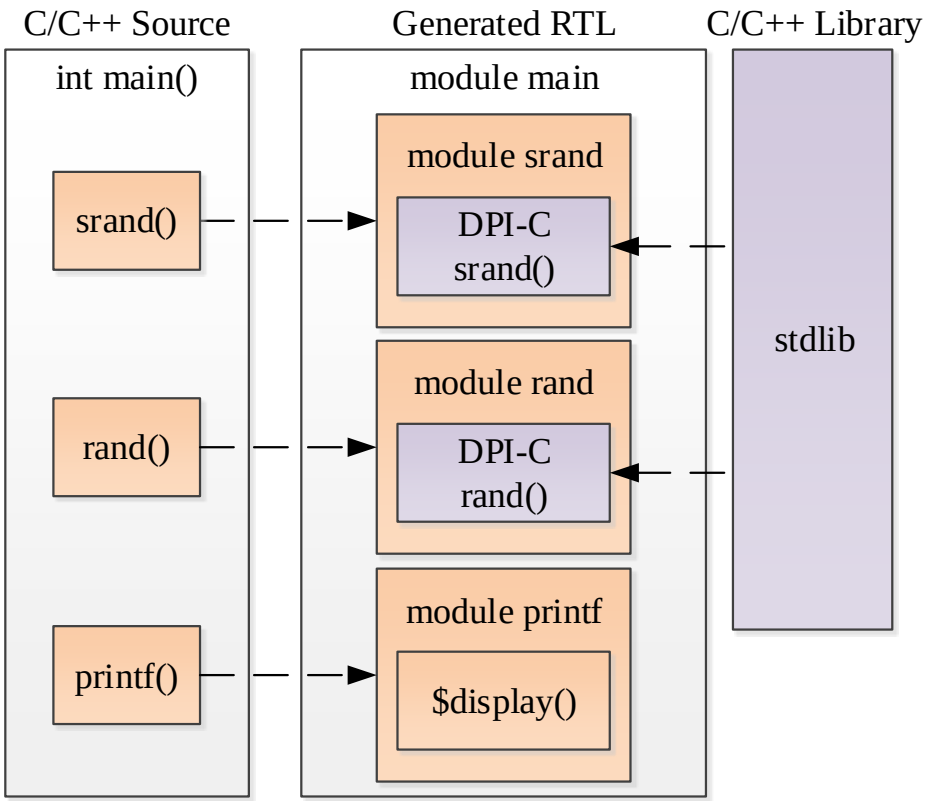

Figure 4.3: Examples of Non-synthesizable IPs

All of these potential uses can be supported transparently; the user is free to use any synthesizable or non-synthesizable Verilog features inside their wrappers.

The HLS process is not concerned with the internal implementation as long as the interface is correctly used.

In this case study, for the applications that use non-synthesizable library functions, we used this ability to verify correct functionality of the applications. The results in Table 4.4 show that, by using the identical software implementation as well as using printing to verify values of internal signals, the non-synthesizable library functions can be supported to deliver more convenient verification process. After verification using non-synthesizable IP cores in the simulation step, HLS users can replace the IP cores with appropriate synthesizable implementations. Alternatively, if the core is used for a debug/verification-only feature, the user can simply wire the 'start' and 'fin' signal together to make the core an empty function call to produce correct RTL without further modification. This feature created by non-synthesizable IPs can significantly reduce the manual effort in partitioning of applications with non-synthesizable functions, so that the partitioning can be much more easily performed, leading to another non-trivial boost in productivity.

4.2.3 Case Study 3: Variable-latency IPs

Fixed-latency IPs allow multiple instantiations while supporting HLS optimizations, but many IPs are variable-latency IPs. These IPs may be large, complex IPs that only require a single instantiation. Each variable latency IP is connected to a waiting state machine that manages IP execution.

Variable latency IPs must internally have control to implement the simple start/fin protocol. This may be as simple as a shift register, or a complex state machine that also manages internal execution states. Many complex protocols such as AXI, Avalon, or Wishbone can be used in a wrapper that connects to this simple protocol.

In this case study, we evaluate the area overhead for supporting variable-latency IPs. For each fixed-latency IP in our benchmark, we create a variable latency alternative. Then, we synthesize the design using all fixed- or all variable-latency IPs and compare synthesis results to demonstrate the overhead. The nn benchmark uses only a single instantiation of the square root function, so it is ideal to compare the area

when that single function is integrated in fixed- vs. variable-latency IPs. Comparing the two designs, the area overhead is 73 ALUTs and 43 registers, which corresponds to less than 2% of the application. This demonstrates that variable latency integration is efficient with very low overhead.

4.2.4 Case Study 4: External Interfaces

Internally instantiated IPs are normally superior to enable HLS core optimizations, but some IPs such as non-synthesizable IPs may be desirable to integrate at the system level in order to better evaluate synthesis results. In both behavioral-level and system-level integration, our automated flow generates testbenches and simulation scripts, but externally integrated IPs partitions IP blocks so that the HLS-generated design can be synthesized after functional verification without further modification.

The automation flow of partitioning of IP blocks is shown in Figure 4.4, where the testbench generation, simulation and synthesis are all automated based on the partitioning result from using external IP interfaces. As an example, synthesizable IPs can be specified as internal IPs on the left part of Figure 4.4 so that RTL specifications can directly instantiate them in the main module, while non-synthesizable IPs can be specified as external IPs on the right part of Figure 4.4 so that they are instantiated in the testbench, externally to the main module. Because of the partitioning information, all ports in main module for external IPs can be generated and connected to corresponding ports in IPs automatically, which eventually helps to realize the automation of testbench and simulation script generation and significantly facilitates functional verification in simulation.

This external interface is also extremely useful to partition and remove unnecessary functions without making major modifications to the source code. For example, in a source that makes heavy use of assertions, the user can simply specify that $assert()$ is an external IP. The HLS engine will partition all calls to use the external IP, and the user can tie together the 'start' and 'fin' signal to disable the assert functionality without making source modifications to individually remove all calls to the $assert()$ function. We use this feature for all non-synthesizable functions so that the

generated core remains synthesizable even if we integrate with non-synthesizable IPs for functional verification. This saves considerable human effort in partitioning for removing non-synthesizable functions for exploring synthesis results, thus introduces significant productivity improvement.

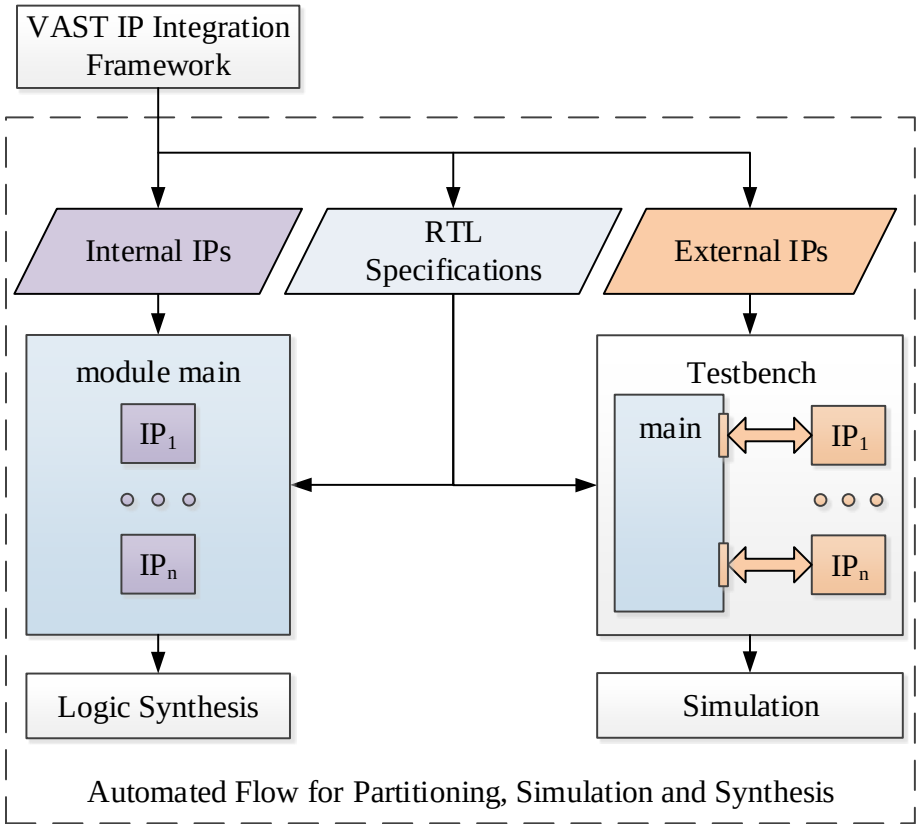

Figure 4.4: Automation Flow of External Interfaces

4.2.5 Other Potential Applications

In addition to the use cases that our applications and case studies explicitly demonstrate, our generalized IP integration technique opens the possibility of a variety of development flows for HLS.

First, flexible IP integration allows hierarchical optimization and design flows. Combination functional units such as multiply-accumulate are common small-scale

optimizations, but developers also commonly start with a small subset of an application and repeatedly expand the scope of computation. With flexible IP integration, a user can first use pre-defined IPs for elementary optimizations and iteratively select common groups of IPs to create specialized merged functional units. Similarly, the user can partition large sub-functions out of the design, verify the control path and then iteratively add and verify sub-functions in order to clearly identify the sources of design bottlenecks.

In addition, flexible IP integration may allow entire specialized design flows. Because our IP integration framework can perform user-specified mapping of any instruction or function to an IP, the HLS user may also map **all** functional units to implementations specialized for domain optimizations such as improved reliability or error tolerance [148].

4.3 Conclusions

In this chapter we demonstrated a generalized IP integration framework integrated with our HLS flow. Using two effective interfaces for fixed- and variable-latency IPs, and support for user-defined mappings between instructions or functions and instantiated IP blocks, we support a wide variety of IP blocks as both internal instantiations and as system-level integrated IP blocks. This generalized IP support opens a wide variety of development flows for HLS including incremental HLS, support for non-synthesizable IP blocks for debug and validation, design partitioning to remove expensive or undesirable sub-functions, as well as more complex flows such as reliability-oriented HLS where every functional unit is replaced by customized components for increased reliability. Compared to related work discussed in Chapter 2, such as [41], our framework does not need to impose extra translation overhead nor need to compromise the optimized performance from RTL IPs by using HLS-generated RTL blocks instead, thus delivering low-overhead design iterations and high-performance IP instantiation for better QoR. As demonstrated by these case studies, our IP integration framework can integrate any user-specified IP sources, such as Altera and Xilinx IP sources, to implement the any specified functions or instructions without

disabling HLS optimizations, thus realizing more reusable RTL IPs being integrated for productivity improvement. In addition, the behavioral-level integration strategy takes advantage of HLS kernel to resolve the challenges of efficiently integrating IPs for exploiting parallelism and partitioning of applications, leading to considerable savings in design effort and refinement iterations compared to manual exploration in system-level strategy. The non-synthesizable integration option also facilitates the prototyping and early verification by providing support to various non-synthesizable library functions. All of these features contribute to productivity improvement. Our IP integration framework is not limited to IPs targeting FPGA platforms, and can be applied to ASIC design flows as well.

Chapter 5

AutoSLIDE

In this chapter, we address the challenges of source-level debugging of HLS-produced designs for productivity improvement. Despite the advantages of automated translation from HLL descriptions to RTL implementations, functional verification of HLS-produced designs remains a major challenge.

Development with HLS tools starts with a traditional software design flow: an algorithm is implemented and verified as functionally correct in software using tools such as GDB before passing to HLS for RTL generation. After the software stage, the source is free of *deterministic* bugs, but software execution can mask *non-deterministic* bugs that will lead to functionally incorrect hardware. For example, subtle bugs activated due to non-deterministic packet arrival orders may be masked in software but produce incorrect hardware.

These *non-deterministic* bugs are typically more challenging to identify; they may be data-dependent or produce no functional incorrectness in software yet produce incorrect hardware. For this reason, there has been a proliferation of works that aid in HLS-produced RTL debug [43, 45, 50, 46, 51, 52, 53, 54, 55]. Although it is possible to debug deterministic bugs as well, it is easier to find such bugs in traditional software tools, and thus users generally only perform RTL debug when software tools are insufficient.

Functional verification and debug of RTL is an increasingly expensive portion

of the hardware design cycle, consuming over 50% of the time in many digital designs [35]. Debug time is particularly important and challenging; when an execution mismatch is discovered, the user must trace backwards to determine the earliest incorrect value and diagnose the source of the problem. This may require detailed tracing of hundreds of signals through thousands of cycles. HLS-produced RTL may exacerbate this problem, with non-human-readable RTL and an unclear relation between input software and RTL statements. Thus, for adoption of HLS tools and acceleration of the design cycle, it is critical for HLS tools to assist in verification and debug.

HLS tools assist through testbench generation and integration with software testbenches [34, 57, 58, 59]. Many academic efforts also assist in automating signal selection and efficient use of trace-buffers for emulation-based verification and debug [53, 54, 55]. In addition, automated correlation [149] from gate-level descriptions to RTL level descriptions assists in diagnosing the low-level discrepancies, while leaving the correlation between the RTL descriptions and HLL descriptions unaddressed, which is the critical factor in the verification and debugging in HLS design flow. Furthermore, recent works have used automated comparison of software and hardware execution to help determine the earliest instance of execution mismatch [60, 50]. However, the first detected execution mismatch is not necessarily the source of a bug; for example, we may detect an execution mismatch at an array out-of-bounds access, but the source is the instruction(s) that set the index variable incorrectly – in a complex, non-human-readable datapath, it is important that HLS tools assist in tracing a bug to its source.

To face this important challenge, in this chapter, we present AutoSLIDE, an automated verification framework for HLS that is able to pinpoint the exact origin of design bugs in the source code. AutoSLIDE instruments the application source code and uses software-generated traces to insert verification code into RTL to assist debug and precisely locate bugs. When simulation detects an execution mismatch, AutoSLIDE backtraces the datapath to determine all intermediate nodes between the mismatch and the most recent verified instruction(s). These nodes are then automatically instrumented and compared for a precise diagnosis of the instruction(s) that are potential bug sources; including the exact line(s) of C/C++ code and, for

complex statements, to precise operations within that line. This automated process substantially reduces debug effort by eliminating the manual effort of precisely determining the location of bugs identified via execution mismatch, thus leading to enhanced debugging efficiency for significant productivity improvement. We demonstrate our verification framework by detecting and localizing real bugs from former versions of the CHStone benchmark suite. We also evaluate efficiency in terms of HLS and software trace gathering overheads, trace size, and simulation time.

This chapter extends our published work [150] in source-level debugging and contributes to debugging and verification for the HLS-based design flow with:

- Automated insertion of verification code to detect non-deterministic bugs
- The first automated precise bug localization tool for HLS that back-traces from an execution mismatch to determine the exact bug source(s) in C/C++
- Substantial reduction in user debug effort due to precise diagnosis of bugs, and bug sources
- A demonstration that AutoSLIDE detects and localizes all bugs with small trace size, reduced simulation time, and low HLS process overhead

The rest of this chapter is organized as follows. Section 5.1 discusses the background techniques, i.e. cross-layer mapping and tracing critical operations, that are used in our debugging framework. Section 5.2 describes the implementation of the AutoSLIDE framework in detail. Section 5.3 demonstrates the effectiveness of the AutoSLIDE framework and evaluates the overhead of this verification technique.

5.1 HLS and Source-Level Debugging

Our source-level instrumentation and debugging framework is built on our in-house VAST HLS framework, as discussed in Chapter 3, which accepts C/C++ source inputs, and produces RTL implementations in Verilog together with an RTL testbench, simulation and synthesis scripts. This HLS framework is an LLVM-based framework, similar to LegUp [138] and Vivado HLS [34], and it uses the LLVM-IR [151]

format during software and hardware optimization passes before producing Verilog RTL. For effective source-level debugging, it is important that we 1) maintain a mapping between the C/C++ source inputs, LLVM-IR operations and the RTL datapath structures, and 2) identify a minimal set of *critical* operations to trace to ensure high coverage with low verification overhead. The mapping provides useful diagnostic information for users to quickly locate the relevant location in the source code, and the low verification overhead helps to keep the additional simulation time as low as possible. Both of them contribute to reducing effort and time for the debugging process, thus leading to increased debugging efficiency and overall productivity improvement. We describe the two requirements for effective source-level debugging in this section.

5.1.1 Cross-Layer Mapping

During the source-code to LLVM-IR translation, a single line of source-code will be translated into one or more LLVM-IR operations. Although many LLVM-IR operations may map to the same line of C/C++ code, the more detailed LLVM-IR precisely determines which operand and/or operator is the underlying cause of a bug, whereas one line of C/C++ source can be complex with many operands and operators.

During HLS, the LLVM-IR will be transformed and optimized repeatedly both during front-end and hardware-specific optimizations. These transformations parallelize, move operations between basic blocks, eliminate redundant operations or duplicate operations for predication. All of these operations transform the sequence or timing of operations while retaining functional equivalence.

In addition, HLS will transform the LLVM-IR to RTL. Similar to the C/C++ to LLVM-IR transformation, a single complex LLVM-IR instruction (e.g. switch) may be mapped to multiple RTL nodes, but sufficiently simple arithmetic/logic LLVM-IR operations (e.g. add, zero extend) tend to be mapped to a single RTL node. However, the HLS process performs resource sharing; thus, although each LLVM-IR operation maps to a single RTL node, an RTL node may be time-shared by many LLVM-IR operations, potentially forming a many-to-many mapping between LLVM-IR and RTL. This introduces extra complexity, as we must not only retain the sequence

of values for each variable/operation, but also the time-interleaving of sequences of values by tracking the FSM state in which each assignment occurs.

5.1.2 Tracing Critical Operations

Prior work in source-level debugging depended either on user-input or iterative debugging to select a subset of signals to examine [53, 54, 55, 43, 45], or exhaustively examining all signals [50]. These solutions are both inadequate, either requiring substantial effort to discover the signals to trace or incurring significant overhead by tracing everything. Instead of exhaustive tracing, it is important to identify the critical set of operations that characterize correct operation of the application. This critical set should be small to reduce overhead, yet provide sufficient bug detection coverage so that any functionally incorrect RTL will be caught by at least one of the critical operations. Furthermore, because of the many-to-many mapping of LLVM-IR to RTL, it is critical that we evaluate operations at clock cycle edges so that we can also reliably use the FSM and clock-cycle timestamp information to ensure correct evaluation of verification code.

Verifying register values for critical operations at clock edges provides full coverage of the application implementation despite not selecting intermediate datapath operations. These datapath nodes can be omitted for a low-overhead bug detection with substantially reduced trace size, and once a bug is identified, we will identify and instrument related datapath operations to gather fine-grained trace data for further analysis.

In this work, we follow an automated two-stage based instrumentation; first, we instrument a sufficient subset of instructions to determine if there is a bug. Then, if a mismatch is found, we automatically determine the appropriate operations to instrument and gather additional trace data for the tree of datapath nodes rooted on the mismatch. This hybrid method balances the overhead benefits of coarse-grained tracing with the bug detection precision of fine-grained tracing.

Our source-level instrumentation and debugging concentrates primarily on assisting with diagnosis of *non-deterministic* bugs, which can trigger non-deterministic

behavior of HLS-generated output and can be particularly challenging to discover. We target the bugs that are activated because they are handled differently in software and hardware due to the non-deterministic characteristic. For example, bugs activated due to non-deterministic uninitialized values or addresses in software, can be handled in a different manner in hardware. Similarly, a bug that leads to an access to a random value in software (e.g. an overrun of an array that leads to an out-of-bound access to an unknown value) can also leads to another unexpected access in hardware, which most likely will trigger a discrepancy between software execution and hardware execution. Such bugs may be data dependent, and only activated in the hardware implementation. Thus, test-vector selection is still an important aspect of the debugging process, but we demonstrate that AutoSLIDE already effectively detects and precisely localizes bugs with typical test vectors. We describe our verification framework in the following section.

In Chapter 3, we described the HLS tool, and the motivation for our AutoSLIDE source-level instrumentation and debugging framework. AutoSLIDE maintains mappings between RTL datapath operations, LLVM-IR operations, and C/C++ source code to precisely pinpoint the root-cause of source-level bugs. This precise localization directly addresses the pain point of debugging, i.e. determine the underlying cause of bugs, thus leading to significant savings in manual backtracing and diagnosis for productivity improvement. Although AutoSLIDE is developed using our in-house HLS framework, this strategy can be applied to other HLS tools as long as the tool retains a mapping between hardware structures and higher-level variables. We will now discuss the implementation of the AutoSLIDE framework in detail.

5.2 AutoSLIDE Framework

An overview of AutoSLIDE is shown in Figure 5.1. AutoSLIDE features a two-stage verification flow (orange) integrated with HLS (light blue), consisting of a coarse-grained stage that instruments critical operations and produces RTL with inserted verification code, data for datapath tracing, and C/C++ correspondence using LLVM source tracing [152]. After the coarse-grained stage, an RTL simulation is performed

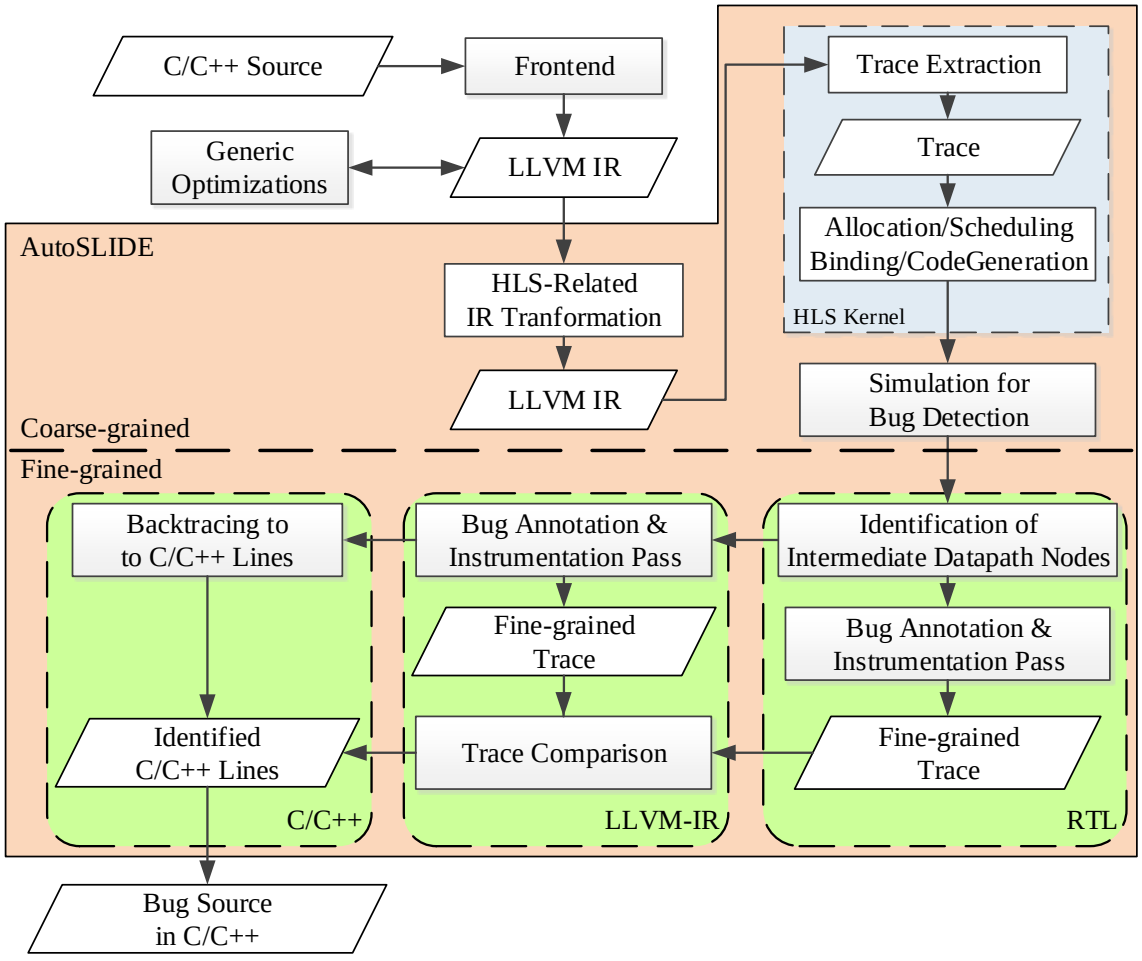

Figure 5.1: Verification Flow in HLS

to determine if there is a bug. Any mismatch exits simulation and passes diagnostic information to the fine-grained stage for analysis.

The location of a mismatch is insufficient to determine the exact source of a bug. The mismatch is simply the location where a bug is detected, but the location where it is introduced may be separated by many lines of code with branching, looping, and function call/returns, or not exist in the same file.

Therefore, to further examine the root-cause of a bug, we must trace backwards through the hardware datapath to identify suspect nodes and gather further instrumentation data for those nodes in both RTL and LLVM-IR. The fine-grained stage

traces the RTL datapath and instruments operations that may be the root-cause; specifically, all operations on the datapath tree between the mismatch and previously verified operations, which will precisely identify the instruction(s) that caused the mismatch. We will now discuss the coarse-grained and fine-grained stages in further detail.

5.2.1 Coarse-grained Stage

Our coarse-grained instrumentation gathers data on load, store, branch, switch, and ϕ operations. ϕ operations are selections: for example, *%indvar = phi i32 [0, %Loop-Header], [%nextindvar, %Loop]* assigns the value of index value to 0 or *%nextindvar* depending on the control-flow – either the first or as subsequent iteration of the loop. Critical operations all correspond to either control- or data-register assignments; thus, although we verify critical operations, when a mismatch is discovered, the root cause may be due to datapath nodes (combinational logic) implicitly evaluated by the verification statement. As discussed in Section 5.1, we also consider resource sharing, the current FSM state, and timestamp as part of verification.

It is important to emphasize that not all registers are instrumented – many intermediate registers for pipelines or intermediate storage do not correspond to the above critical instructions that affect the computation datapath. For data operands of critical instructions, we can simply record the data values from software for comparison to run-time hardware values. However, for address operands, we must perform some additional translation.

Address variables, such as the operand *%ptr* in *%val = load i32* %ptr* are an important part of verification to detect access to undefined or uninitialized values, and out-of-bounds array accesses. Internal memory structures will be statically allocated, but the address variables from a software trace will implicitly include a software-specific sub-field of bits from the memory map, which are not relevant for mismatch. We cannot simply mask these bits, as that would incorrectly make all address values valid instead of detecting incorrect addresses. Thus, for each of the application's internal memories, we create a mapping between the software memory

map and an internal memory map (for verification). The internal memory map contains the information about the layout of the memory block allocated to implement the corresponding software memory block. Together with static information about the allocation size, we thus determine whether an address is valid.

Once we have identified and annotated all critical operations, we use LLVM execution engine to gather the trace data. This trace data is then used to generate verification blocks that compare the expected (software) value to the simulation run-time value of variables. Due to HLS operations such as binding, the same physical hardware structure may be used for multiple different RTL variables throughout simulation run-time. Thus, the verification code is not simply a sequence of values, but also includes verification that the value changes in the correct state of the FSM, and the sequence of FSM states is also correct.

The verification code in RTL terminates the simulation on the first execution mismatch and prints out several sets of diagnostic information. This information can be used directly, but it requires user understanding of the RTL datapath and detailed tracing for bug diagnosis. Thus, it is passed to the fine-grained stage for detailed instrumentation and analysis of the datapath nodes. Diagnostic information includes:

1. Expected value vs. Actual value of operation
2. Timestamp of operation
3. RTL register, LLVM-IR instruction, and C/C++ line of code of operation
4. FSM State

This diagnostic information will be used by the fine-grained stage to determine the range of datapath operations that could be the source of the bug, which will guide the detailed instrumentation process.

5.2.2 Fine-grained Stage

The fine-grained stage receives diagnostic information on a detected bug from the coarse-grained stage. First, it determines the responsible LLVM-IR instruction using

the FSM state to disambiguate possible resource sharing. From a mismatch, we backtrace along the dependence path of each of the operator's operands iteratively to produce a tree with the mismatch as root and prior-verified operations as leaves. The prior-verified operations are identified by backtracing from the root to find the last operations that correspond to critical instructions that have been verified in the coarse-grained stage, as demonstrated later in Algorithm 1. In this tree of operations, the bug activation occurred **after** the leaves, but **before** the verification time-step of the mismatch. The depth of the tree is determined by the root and leaves, and is usually small because the critical instructions verified in coarse-grained are regularly used in the datapath.

The fine-grained stage instruments each node in the tree in RTL and LLVM-IR; in RTL, this is simply printing out the intermediate variables and their value at the timestep of the mismatch. Intermediate nodes are combinational logic that may take multiple values before stabilizing; thus, it is important to print the value at the mismatch timestep to ensure that we gather the data used to compute the mismatched operation, not transient values.

In LLVM-IR, we add fine-grained instrumentation for each instruction in the bug's datapath tree. This fine-grained instrumentation avoids high trace overhead by only tracing the bug's datapath tree instead of exhaustively tracing every instruction. Critical operations are used heavily for memory accesses, and control-flow. Thus, effective coarse-grained instrumentation limits the distance in operations between verified nodes, and also limits the size of potential bug datapath trees. Thus, when paired with the coarse-grained stage, the number of instrumented instructions in the fine-grained stage is usually small and we can gather the trace information quickly. Although typically small, the automated instrumentation also enables instrumentation of a large number of intermediate nodes if necessary for a particularly complex application bug. These two automated instrumentation stages lead to considerable savings compared to manually determining the relevant datapath and instrumenting all the nodes in different layers, thus achieving significant debugging productivity improvement.

To illustrate how the bug datapath tree is identified, we use a real bug example

from the *mips* benchmark from CHStone Version-1.11 [153] in Figure 5.2. For simplicity, we show the RTL and LLVM-IR representations of the benchmark, but each LLVM-IR instruction can be related to a single line of C/C++ code using LLVM source tracing [152]. In this benchmark, the application compares the first eight items in ram1 with the items in rom2 and counts the number of mismatched pairs between the two memories. However, during simulation, we find a mismatch on the 9th iteration of updating the variable stored in reg3. This mismatch corresponds to the line of C code: $main\_result+ = (dmem[j]! = A[j])$;

However, this line of code has a datapath tree including an addition, comparison, and two loads from the respective memories. The bug itself is a discrepancy between the "expected" value to be stored (1), and the actual value stored (0), but the root cause of that bug must be traced through the datapath. From the mismatch, we perform a depth-first search to identify all the nodes in the bug datapath tree, as shown in Algorithm 1. After finding the datapath tree in RTL, we directly build the corresponding LLVM-IR tree using the RTL to LLVM-IR correspondence. For each node in this datapath tree, we perform additional instrumentation in both RTL and LLVM-IR within the same simulation run-time.

Through comparison of the fine-grained trace data, we can identify that the bug comes from the portion of the datapath tree related to rom2 in the RTL and const array2 in the LLVM-IR (outlined in blue in Figure 5.2). Backtracing along this datapath identifies that the source array only has eight elements; in software, the 9th iteration loads an undefined value (*load2 = 2409889792* as shown Figure 5.3), whereas the hardware rolls over and accesses the element at address 0 again (*reg2_o = 22*). This bug datapath thus identifies the array declaration, loop index *j*, and array reference lines of code as being involved in the bug. Although this still requires user knowledge to determine how to fix the bug (i.e. change the allocation size, reduce the number of loop iterations, or make the index computation modulo 8), this correctly identifies not just the location of a mismatch, but the C/C++ code creating the mismatch.

Although the mismatch statement included a comparison, addition, and memory access to ram1, AutoSLIDE can correctly filter that (in this case) the access to ram1 is

Algorithm 1: Algorithm of Instrumentation of Datapaths

```
typedef OperandList::op_iterator ChildIt;
set<DatapathNode *>&Visited;
vector<pair<DatapathNode *, ChildIt>>VisitStack;
DatapathNode *Root = this;
VisitStack.push_back(make_pair(Root, op_begin()));
while !VisitStack.empty() do
    DatapathNode *Node = VisitStack.back().first;
    ChildIt It = VisitStack.back().second;
    // print current node if all children are visited
    if It == Node->op_end() then
        VisitStack.pop_back();
        print(Node->Value);
        continue;
    end
    // Otherwise, remember this node and visit its children first
    DatapathNode *ChildNode = It->get(); ++VisitStack.back().second;
    if ChildNode is not the last verified register then
        // If this ChildNode is already visted
        if !Visited.insert(ChildExpr).second then
            continue;
        end
        VisitStack.push_back(pair(ChildNode, ChildNode->op_begin()));
    end
end
```

not related to the bug path since it was verified. Through fine-grained comparison of datapath nodes along the suspect path to the source of bug, AutoSLIDE can pinpoint the exact operations and C/C++ lines, including identifying which operands or sub-expressions in a complex compositional statement are related (or unrelated) to the bug datapath.

In summary, to quickly detect and diagnose bugs, AutoSLIDE features a two-stage method with low-overhead coarse-grained instrumentation and simulation to quickly identify whether a bug exists, followed by fine-grained instrumentation and datapath analysis to precisely diagnose which lines of code are the underlying cause. These two stages can be finished in one simulation. This allows the AutoSLIDE framework to detect and diagnose bugs with low latency (in simulated clock cycles), and with low-overhead in simulation time and RTL generation. The precise diagnosis of RTL operations, LLVM-IR operations and lines of C/C++ source code automates

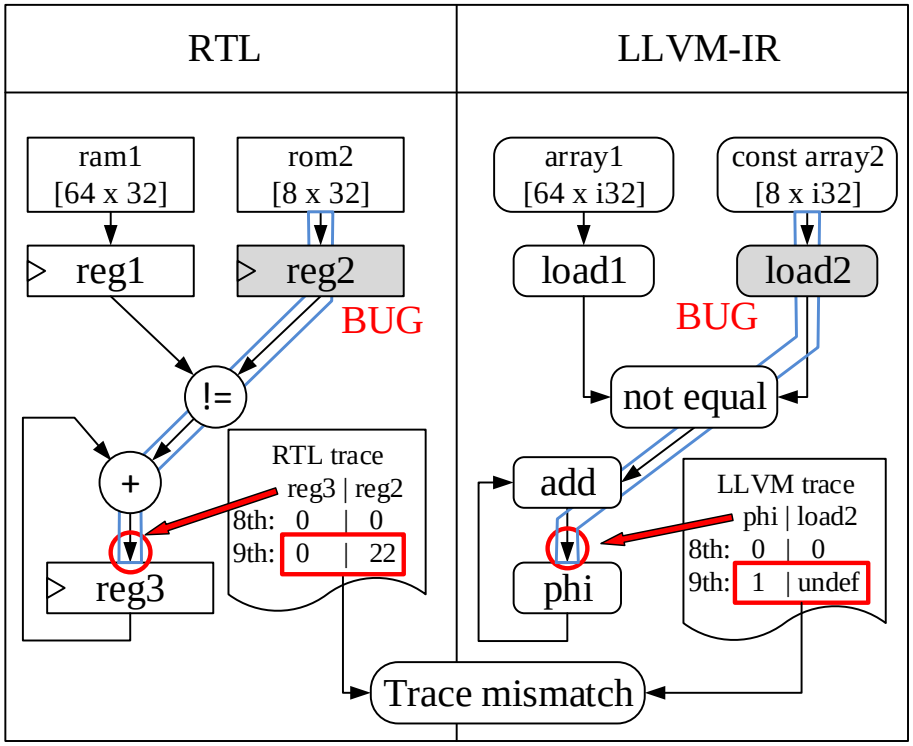

Figure 5.2: Instrumentation of Intermediate Datapath Nodes

detection and diagnosis of bugs, leaving only determination of how to fix the bug to the user. The automated instrumentation, detection and diagnosis lead to considerable savings compared to manually determining the relevant datapath and instrumenting all the nodes in different layers, thus achieving significant debugging productivity improvement.

5.3 Results

We now study the effectiveness of AutoSLIDE using the set of bugs from the CHStone suite [153] used in HQED [70]. For each source-level bug, we isolate the bug by patching all known bugs in the version of the benchmark, then reintroducing each bug one at a time. If there are multiple bugs in the same benchmark, we differentiate the bugs by their order in the benchmark source. As discussed earlier, we concentrate on

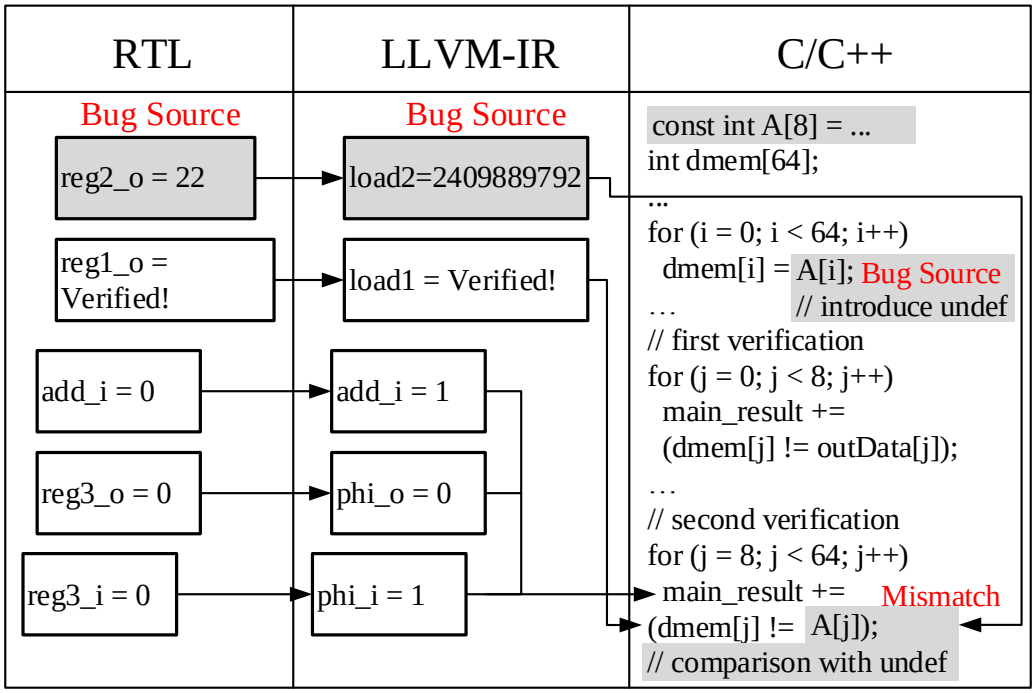

Figure 5.3: Printed Values of Intermediate Datapath Nodes

non-deterministic bugs, which can be subtle interactions related to time-interleaved transactions, or undefined behavior due to uninitialized variables or out-of-bound accesses.

For each bug, we execute the AutoSLIDE framework to determine the location of an initial bug mismatch, and identify the datapath nodes and locate the lines of source. It is important to emphasize that AutoSLIDE has no apriori information about existence of a bug, type of bug, or locations of errors.

5.3.1 Bug Detection and Backtracing

First, we evaluate each of the source level bugs to determine the effectiveness of AutoSLIDE. As expected, AutoSLIDE detects the existence of each bug in the coarse-grained stage with low latency: two of the bugs in the exact cycle that the bug is activated, and the other three one cycle after activation (Table 5.1). The verification RTL is also printed, with values bolded. The coarse-grained stage alone is sufficient

to detect the **existence** of bugs quickly.

However, as shown in Table 5.2, the location of the mismatch does not correspond to the location(s) in the bug datapath tree that contribute to the actual bug. The line of source where the mismatch is detected (in red) corresponds to a critical operation, but the actual bug from the C/C++ source code (bold and highlighted in yellow) may be due to other potential statements such as uninitialized variables, loop bounds, or more subtle interactions. Furthermore, we can see that this datapath tree in the source can be complex: in gsm, the datapath tree crosses function calls and several different source files. Even so, we see that the fine-grained stage is able to back-trace the datapath and automatically determine the root-cause of the bug in RTL and C/C++.

In the gsm bug, the fine-grained pass identifies the loop bound and array declaration for a data array passed to a sub-function. With this detailed information, the user can determine how to solve the bug – change loop bounds, change $k >= 0$ to $k > 0$, or increase the declaration size. Although the other four examples are similar, AutoSLIDE can also detect more subtle bugs such as the logic bugs discussed in QED [154]. These detection results show that AutoSLIDE directly addresses the pain point of debugging, i.e. diagnosing and localizing the root cause of a bug, which

Table 5.1: Source-Level Bug Results

Benchmark/ Bug Locations	**Bug Description**	**AutoSLIDE**		
		Detection Latency	**Detection**	**RTL mismatched node(bold)**
gsm/ lpc.c 156-158	Out-of-bounds access	0	Violation of end address of a global array	if(RamAddr[] >= **EndAddr**) $finish();
mips(bug1)/ mips.c 102	Read of uninitialized variable	1	Mismatch between undefined and RTL default value	if(32'h3B7C9F8 != **Hi_0**) $finish();
mips(bug2)/ mips.c 132-135	Read off the end of array	1	Mismatch between out-of-bound access and rolling access	if(32'h1 != **main_result_add**) $finish();
motion(bug1)/ mpeg2.c 225-226	Read off the end of array	0	Mismatch caused by pointer incremented off the end-of-array	if(8'h38 != **ld_Rdptr_deref**) $finish();
motion(bug2)/ getbits.c 144, 155, 160	Out-of-bounds shifts	1	Mismatch of undefined value from out-of-bounds access	if(64'h2D != **PMV_0_0_0**) $finish();

Table 5.2: **Bug Sources** vs. **Mismatches**

gsm
gsm.c 29 : #define **N 160** 94 : **so[N]** ; 95 : word LARc[M]; 101: Gsm_LPC_Analysis (**so**, LARc); ... lpc.c 314: Gsm_LPC_Analysis (word *s, word *LARc) { ... 157: for (**k = 160** ; k >= 0; k--) 158: *s++ <<= scalauto;
mips(bug1)
mips.c 102: **Hi** ; ... 179: reg[rd] = Hi;
mips(bug2)
mips.c 91 : const int **A[8]** = { 22, ... 132: for (i = 0; i < **64** ; i++) 134: dmem[i] = **A[i]** ; ... 303: **main_result** += (dmem[j] != **A[j]**);
motion(bug1)
global.h 75 : unsigned char **ld_Rdbfr**[2048] ; mpeg2.c 39 : #define **Num 2048** 195: **ld_Rdptr** = **ld_Rdbfr** + 2048; 225: for (i = 0; i < **Num** ; i++) 226: main_result+=(***ld_Rdptr**++!=out_ld_Rdptr[i]);
motion(bug2)
mpeg2.c 351: int **PMV** [2][2][2]; motion.c 87 : motion_vector(**PMV** [0][s], ... 115: decode_motion_vector (& **PMV** [0], ... 133: if (mvscale) 134: **PMV**[1] <<= 1;

can be very time-consuming during debugging process. In addition, AutoSLIDE accelerates the localization process not only by automating the backtracing, but also

by efficiently narrowing down the relevant datapath to a much smaller range, which significantly reduces the overhead of backtracing and speeds up the verification and debugging process.

5.3.2 Overhead of our Verification Technique

AutoSLIDE uses RTL simulation, correlation with software traces, and automated backtracing; although the precise detection presents a substantial reduction in effort compared to manual bug tracing, simulation may be expensive compared to emulation-based techniques. Thus, we also demonstrate that AutoSLIDE's two-stage instrumentation technique effectively minimizes overhead in HLS process time and simulation time.

We compare the HLS run-time for each benchmark with and without the addition of the AutoSLIDE instrumentation. The overhead of instrumenting the LLVM-IR, gathering traces and producing extra verification code represents an average 27% HLS run-time overhead, primarily due to the trace gathering, which is a software execution with higher file I/O to gather all intermediate values of the instrumented variables. Although significant, this overhead is a once-per-bug overhead, as the framework eliminates the need for iterative selection and evaluation of trace signals given a good set of input test vectors.

Because the software execution for trace gathering is the primary source of HLS overhead, we additionally evaluate the overhead of trace gathering compared to normal software execution. The set of instrumented critical instructions is small, and thus the run-time overhead of trace gathering is only 10% on average, with negligible overhead during the fine-grained instrumentation. In comparison, an approach that starts with exhaustive (fine-grained) instrumentation would incur significant overheads. Exhaustive instrumentation has 300% overhead compared to AutoSLIDE (3× slower than AutoSLIDE), with the coarse-grained traces as much as 8× smaller than exhaustive, and the corresponding fine-grained traces over 100× smaller than the exhaustive traces as shown in Table 5.3.

The AutoSLIDE framework is a pre-synthesis verification technique, so it does not

Table 5.3: Comparison of Trace Size in Number of Values

Benchmarks	gsm	mips (bug1)	mips (bug2)	motion (bug1)	motion (bug2)
Exhaustive	53670	16541	16778	23320	4873
Coarse-Grained	8166	5624	6352	12501	4305
Fine-Grained	161	1221	3	2048	1

incur any area overhead – all verification blocks are implemented in SYNTHESIS OFF regions, and automatically ignored during logic synthesis. AutoSLIDE is intended primarily for debugging, so it is expected that a user would generate RTL without verification code once the design is verified to be bug-free.

The run-time of simulation also shows that complex verification code can substantially increase total simulation time of **functionally correct** RTL – on average 58%, but as high as 2.5× longer simulation. However, AutoSLIDE should be used to diagnose bugs in functionally incorrect code. Although the overhead when simulating correct RTL can be substantial, the early exit (with detailed diagnostic information) when a bug is detected reduces simulation time in practice, with an average speedup of 1.6× over a full simulation without the AutoSLIDE framework. More importantly, even though a full simulation (or emulation) can detect the existence of bug(s) by verifying the return value(s), there would be substantial manual effort to backtrace and identify the bug source. Manual back-tracing may require hundreds of signals through thousands of cycles, whereas AutoSLIDE can automatically and precisely localize bugs. Therefore, the speedup in simulation time, the automated tracing and precise localization achieve considerable savings in manual debugging effort, leading to significant overall productivity improvement.

Simulation overhead is directly related to benchmark complexity and the number of critical operations with verification code. If we instead choose to instrument fewer operations, there will be lower RTL simulation overhead at the cost of larger bug datapath trees and thus larger fine-grained traces. We leave the exploration of tuning the number and selection of critical operations compared to the size of the

bug datapath tree for future work.

5.4 Conclusions

We presented AutoSLIDE, an automated verification framework for HLS that is able to pinpoint the exact origin of design bugs in the source code. AutoSLIDE instruments applications, and uses software-generated traces to insert verification code into HLS-produced RTL. On an execution mismatch, the framework backtraces the datapath to automatically instrument intermediate nodes for a more precise diagnosis of the instruction(s) in C/C++ causing the execution mismatch. We demonstrate our framework by detecting and precisely localizing bugs from former versions of the CHStone benchmark suite. This automated process considerably eliminates the manual effort of precisely localizing bug sources related to HW/SW execution mismatch, substantially enhancing the debugging efficiency and improving productivity. Furthermore, we demonstrate the efficiency of our framework with low HLS run-time and simulation time overhead, as well as reductions in trace size and trace gathering time compared to exhaustive approaches. These savings in HLS run-time and simulation time accelerate the verification process, thus also contributing to productivity improvement.

Chapter 6

Automated Trace-Based Verification

Improvement of HLS optimizations indirectly increases productivity through reduced user effort and fewer design iterations for designers to more quickly meet QoR objectives. However, advanced optimizations are challenging to implement and deploy – and challenges in verifying the correctness of optimizations can delay delivery of key optimization advances. In this chapter, we address the challenges related to the verification of HLS tools.

As discussed in Chapter 1, HLS tool development is a huge software development effort, and a significant amount of effort is spent on verification. People use software debugging tools, such as GDB and Vagrind, to detect the run-time incorrectness within in the HLS tools. However, verification of HLS tools is different from typical verification of whether large-scale software systems can be executed to a normal completion. A completion of the execution of HLS tool without throwing any errors or warnings is not equivalent to a correct generation of RTL implementation. Therefore, HLS-generated RTL must be subsequently verified through functional verification to prove that the HLS tool is functionally correct. Because HLS optimizations involve many complex transformations, the verifiability of the correctness of output is further exacerbated. Thus, the functional behavior of the HLS-generated RTL still remains as a major challenge.

There are many works that try to formally verify the functional equivalence between the input specifications in HLL descriptions and output design in RTL descriptions to guarantee that the HLS-generated output exactly follows the behavioral specifications in HLL. However, formal verification relies on time-consuming solving process for the satisfiability, resulting in exponentially increased processing time overhead for large and complex designs.

In addition, HLS tools are built on underlying compiler infrastructures, such as LLVM [26]. Despite the effort in the compiler community for a fully formally verified compiler, the current underlying infrastructure is not fully formally-verified. Because these aggressive non-formally-verified compiler optimizations are important for generating IR with good QoR, and the QoR of input IR has significant impact on the QoR of HLS-produced output, these compiler optimizations are critical and necessary for overall QoR. However, because of the aggressive, non-formally-verified compiler infrastructure, it is difficult to infer whether the HLS tool implements the HLL behavioral description in a formally valid way for each operation in HLS optimizations. Therefore, it represents a challenge in the verification of the correctness of HLS transformation from the HLL source to RTL, and also a challenge to use formal methods to verify the whole HLS process, in which non-formally-verified transformations are involved.

In addition to those formal methods, simulation-based verification uses real test vectors to verify the functionality, so that non-formally-verified optimizations can be accepted in the test flow. However, traditional simulation-based approaches may only find the discrepancy between the HLL and RTL descriptions at the end of the simulation, which may be millions of cycles away from where the bug is introduced. The large detection latency not only increases the simulation time significantly, but also imposes huge effort on users to correctly select and monitor a large set of relevant signals and backtrace them for millions of cycles. Furthermore, HLS-produced RTL is typically not intended to be human-readable; the RTL may be less intuitive to read and correlate to expected behavior than debug of manually-produced RTL. As capacity and complexity of designs grow, this traditional simulation-based method becomes impractical for large designs, resulting in a challenge in simulation-based

verification. Therefore, it is critical to overcome these challenges to deliver both useful diagnostic information and reduced simulation time.

The HLS process performs many transformations to parallelize and optimize execution; thus, we cannot validate application correctness by comparing the exact order of operations. However, we can fundamentally characterize correct execution with a few properties: input data received, output data produced, conditional control transitions, correct propagation of data through data selection (ϕ-node) operations, and forward progress in execution.

In this chapter, we present a framework to debug and validate all operations that characterize the behavior of an application. We use trace-generation to generate the set of expected values for all verified operation types and automatically insert RTL verification code for each operation and value pair, together with information about the correspondence between the RTL and operation in LLVM-IR. To demonstrate the effectiveness of our verification framework, we create an infrastructure to randomly insert bugs in the HLS-generated RTL, validate that our verification framework detects the bug, and compute the latency of bug detection. we demonstrate that our verification framework detects 94.8% of the 12000 randomly inserted bugs, with over 78% of the bugs detected in less than 10 cycles. The remaining undetected bugs are not activated by the default test vectors: additional test vectors could detect these bugs as well. Similar to other HLS flows, our verification framework also autogenerates RTL testbenches and simulation scripts to speedup overall verification time for further productivity improvement.

This work contributes to debugging and verification of HLS tools with:

- A trace-based approach to automatically insert RTL verification code for all operations that characterize correct application execution.

- An intermediate execution that automatically gathers expected value information for all characteristic operations.

- A random bug insertion evaluation infrastructure to insert random RTL bugs and validate that the verification framework correctly detects the bug.

- A demonstration that our verification technique detects 78% of bugs with latency less than 10 cycles.

- A demonstration that our verification framework adds only 25% overhead to the HLS process and 58% overhead to RTL simulation (in non-faulty simulations). Due to early exit on bug detection, bugs are on average detected with 50% lower latency than a functionally correct simulation.

The rest of the chapter is organized as follows: Section 6.1 provides an overview of the verification in our VAST HLS framework; Section 6.2 presents details of our verification framework including generation of software traces and RTL verification code; Section 6.3 discusses our experimental setup and results and finally, we present our conclusions in Section 6.4.

6.1 VAST HLS and RTL Verification

Our trace-based verification framework is built on our VAST HLS framework, as discussed in Chapter 3. VAST accepts C/C++ source inputs, and produces RTL implementations in Verilog together with an RTL testbench, simulation and synthesis scripts. It generates a hardware-oriented intermediate representation for hardware-specific optimization passes; VAST keeps track of correspondence between LLVM-IR instructions and VAST-IR operations, which is important for our purposes in trace-based verification. In addition to our verification framework for this chapter, VAST includes a large number of source code assertions that validate input source code and check for known potential problems in the HLS core. These assertions catch many potential problems before RTL verification, but an HLS developer cannot predict all possible potential bugs.

During the HLS process, the LLVM-IR (and VAST-IR) are repeatedly transformed to optimize the area and latency of the generated hardware design. These optimizations may parallelize operations, move operations between basic blocks, eliminate redundant operations or duplicate operations for improved predication, or transform

operations through constant propagation and strength reduction. All of these optimizations may transform the sequence or timing of operations while retaining functional equivalence. However, in order to verify RTL correctness, we must characterize correct behavior of the hardware independent of these valid (and necessary) performance optimizations.

Therefore, given the reference information collected from software execution, we characterize correct operation of an application with the following high-level invariants:

- For a given reference address and input data set, the value of a load is constant.
- For a given reference address, the value of an output store is known.
- For a given ordering of loop iterations (specified by the input), the sequence of load and store addresses is known.
- For a given ordering of loop iterations, the sequence of loop index values is known.
- For a given program, the set of valid control transfers (branch targets) is known.
- For a given program, a global worst case execution latency (given assumptions on worst case I/O latency) is known.

In addition, we can also verify some hardware specific invariants that are independent of the application information:

- For any shared resource (e.g. memory bank), no two operations sharing the resource may be activated by the control FSM at the same time
- The control FSM must always be in a valid state, and should never transition to an idle state without properly signaling computation completion.

The first two invariants, which related to load/store instructions, guarantee that any incorrect data related to memory access can be detected. The loop-related invariants guarantee that if any incorrectness is produced in the order, or the sequence

of index values, or even the underlying control-flow transitions, then the incorrectness will be caught. In addition, the invariants related to control transfers guarantee that the HLS-produced hardware implementation follow the correct execution flow that matches the software specification. As long as all these invariants are verified, it denotes that the operations of application accept input and generate output values as expected, thus the functionality is guaranteed to be correct. By using all of these invariants that examine functionally critical values in both data-flow and control-flow, we can guarantee that a functionally incorrect RTL output cannot execute without violating at least one of the expected behaviors, with one exception. The only functional RTL bugs that our framework cannot detect are the ones that will produce a correct output despite a bug. For example, if an RTL bug replaces a multiplication operation with an addition, our verification framework will not detect a bug with a test vector with both input operands as 2, because both operations produce a *correct* output of 4. For this reason, it is still important to select test vectors and ensure that the selected set of vectors covers appropriate execution cases; our framework would still detect this bug with a different set of input vectors.

Thus with the exception of such rare cases, with information on the correspondence between RTL operations and LLVM-IR operations, the exact failing verification statement can be used to diagnose the specific underlying problem. Additional information that verifies the sequence of loop iterations, index values, and control transitions serves to verify internal RTL core behavior.

Using these invariants, we can statically determine the set of instructions that must be verified to guarantee functional correctness and full coverage of the application's RTL implementation. Specifically, if we generate verification code for all load, store, branch, switch and ϕ[1] instructions, then the behavior of all critical operations will be verified. Verification code for the hardware-specific invariants will provide additional diagnostic information about potential causes of mismatched execution.

For the hardware-specific invariants, we consider that VAST HLS produces its

[1] ϕ operations represent a selection operation, where the value assigned depends on control flow. ϕ nodes are used to support static-single assignment form data dependence analysis for loop-carry and cross basic-block conditional data dependences.

control FSM in a one-hot format[2], but multiple states may be simultaneously active in the case of global code motion (to simultaneously execute multiple basic blocks), or pipelining (to simultaneously execute multiple iterations of the same basic block). Each state in the FSM corresponds to a single bit; we can verify that no two operations sharing a resource are active at the same time by ensuring that no two assigned FSM states of sharing operations are simultaneously active. Similarly, we can validate control transitions by ensuring next-active states are in the valid set of control transitions for branch and switch statements, and that the FSM is always in a valid state by logically-oring all FSM states (if no state is active, the FSM has unexpectedly halted).

6.2 Trace-Based Verification Framework

The previous section discussed the set of instructions we must verify in order to guarantee coverage of the generated RTL. Each of these instructions may execute many times during dynamic program execution; verification code must verify every value produced by each execution time of instructions so that all the values produced can be covered for the functional verification of HLS tool. Our trace-based verification framework uses LLVM's compilation and execution [155] framework to generate traces of the sequence of input and output values for all instructions of the critical types. Then, using those traces, during the HLS process, RTL code generation also inserts verification code for each traced value of each instruction.

Our verification flow is integrated into the existing VAST HLS flow as shown in Figure 6.1. The new components for this chapter are shown in orange, where we require integration with the steps in blue to provide correspondence between hardware structures and the instrumented LLVM-IR instructions to be verified. VAST HLS uses Clang to parse C/C++ code into LLVM-IR and perform HLS-independent optimizations, followed by HLS-related IR transformations. From this IR, we perform

[2]Grey-coded state machines could also verify the hardware-specific invariants, but one-hot simplifies our verification statements

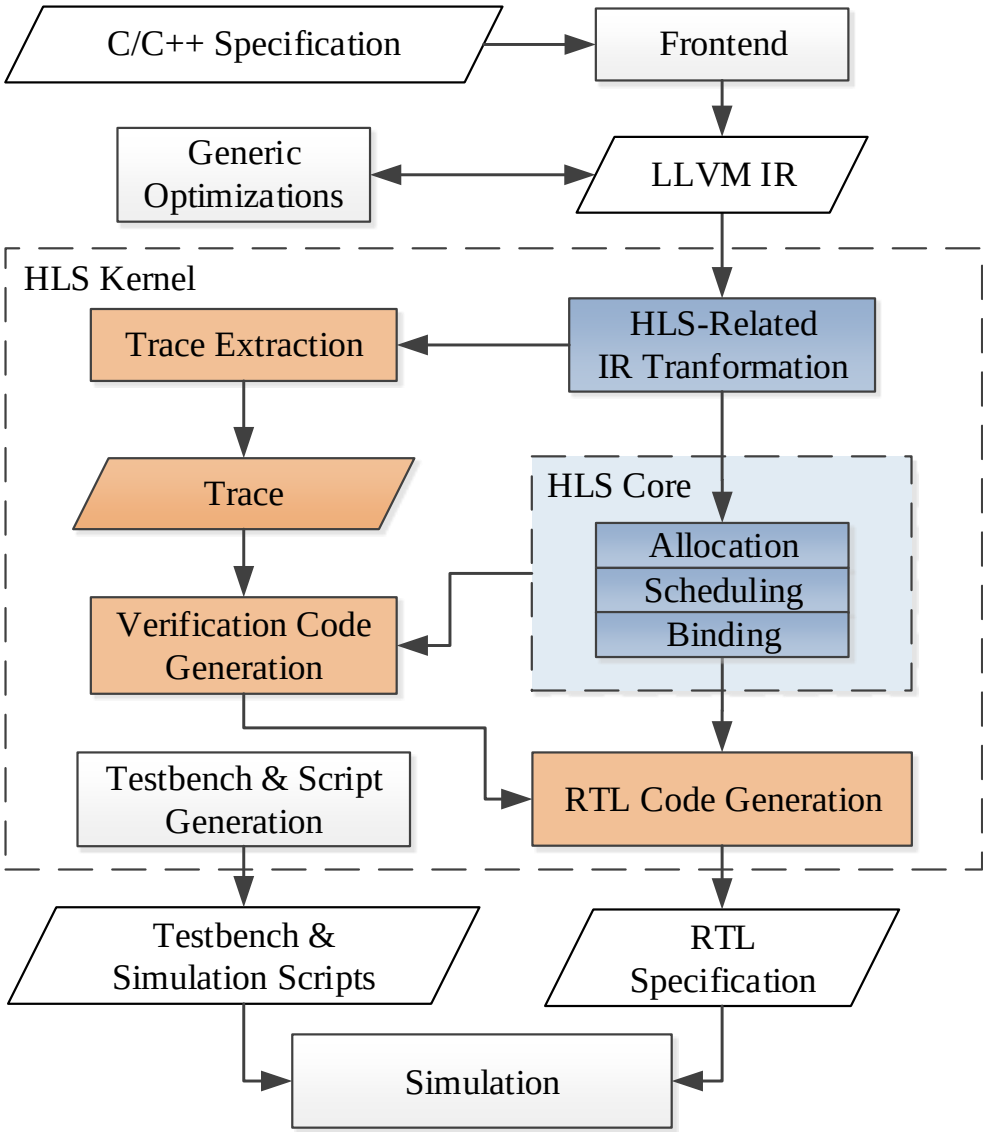

Figure 6.1: Verification Flow in VAST HLS

instrumentation of each targeted instruction type, and perform compilation and execution to generate traces for each instruction. Then, during code generation, we track correspondence between hardware structures and the LLVM-IR instructions; for each hardware structure with trace data, we also generate verification code. Though we have integrated our verification flow on a specific HLS tool, the underlying concept can be applied to other HLS tools. Given that an HLS tool has inbuilt data structures

that can retain the mapping between hardware structures and higher-level variables (LLVM-IR instructions in our case), the HLS code generation can be instrumented to generate verification code blocks for each register using trace data.

In addition to autogeneration of verification code, by using the information about the schedule and interface of HLS-generated designs, VAST HLS also automates RTL testbench generation and simulation scripts; thus, our inserted verification code is automatically used during functional verification. We will now discuss the implementation of these features in detail.

6.2.1 Trace Extraction

Given an input LLVM-IR application, we need to extract traces of sequences of input and output values for each instruction as shown in Figure 6.2. The LLVM infrastructure provides support for compilation and execution of LLVM-IR modules; in our case, we use this functionality to automatically instrument the LLVM-IR and produce a trace for each desired instruction.

For our set of instructions, we will need to verify both address and data values. Furthermore, the data values may have a variety of data types, and bit-level optimizations may produce a variety of (non-power of 2) operand widths. Thus, before recording values, we reformat instruction arguments to perform pointer extraction, value alignment and masking for the relevant bits to compare. Then, the formatted instruction arguments are recorded to an instruction trace using a call to our custom trace-collection function. Example pseudo-code for instruction instrumentation is shown in Figure 6.3. The trace-collection function simply keeps an individual sequence of values for every instrumented LLVM-IR instruction.

After instrumentation of every desired LLVM-IR instruction, we create an LLVM execution engine that executes our instrumented module (which generates the trace data). The current CHStone benchmarks do not require command line parameters, but the execution engine can provide command line parameters if required. In that case, the user is responsible for providing identical input data to the trace-gathering execution engine and the functional verification in RTL. In this chapter, we examine

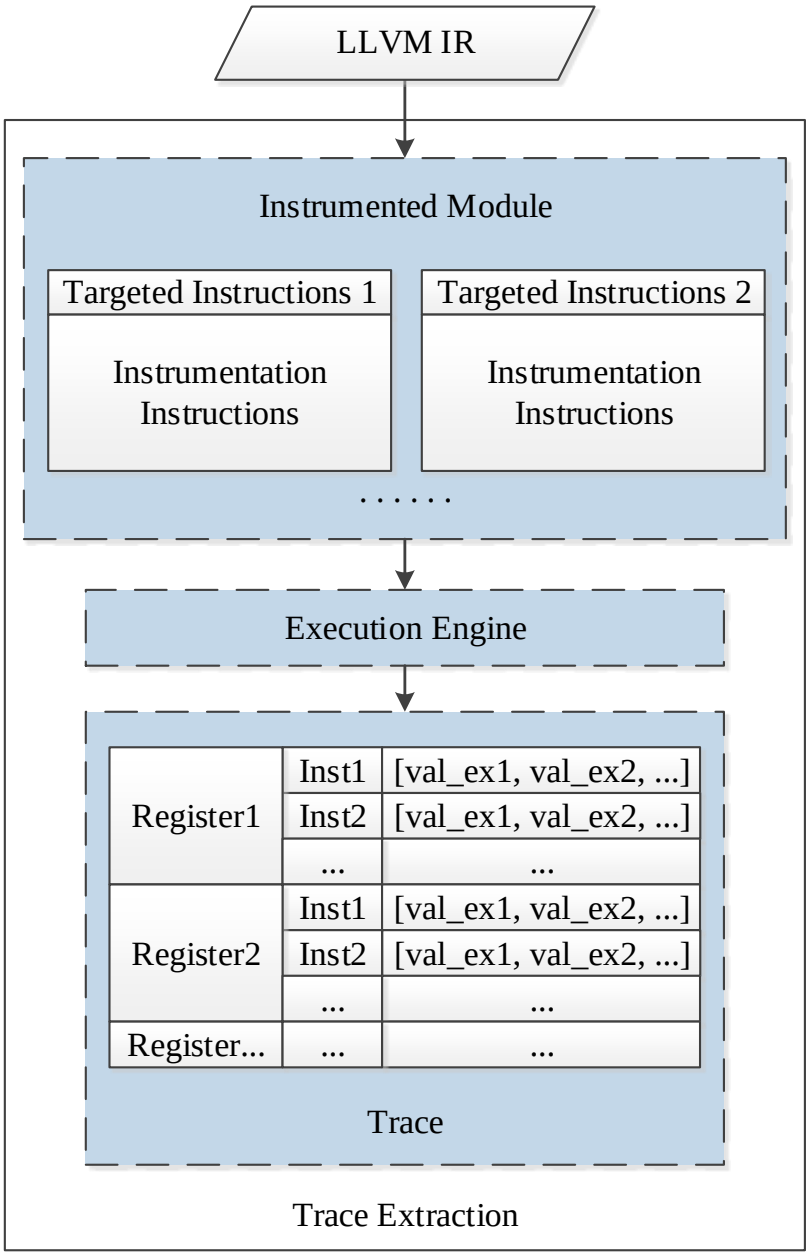

Figure 6.2: Trace Extraction

the CHStone benchmarks with a single set of input data; to detect any data-dependent bugs, the user should use multiple sets of input data: each with uniquely generated verification code. Although in this work, we automatically generate the traces from

within the HLS core, this technique would also be compatible with using properly organized trace input files, e.g. a file with traces recorded in a format that is easy to compare with. However, these files may be large, and maintaining correspondence between internal variables in HLS and the value traces may be challenging.

It is important to note that the execution engine can only generate traces when it can resolve all function bindings so that the values of all inputs and outputs are known. All functions in all CHStone benchmarks are resolved, but if an application uses a library or function without linking a function implementation, the execution engine will be unable to generate traces for any data-dependent instructions.

The execution engine will produce trace data as shown in Figure 6.2. Because of their importance for input/output and control flow, all of the verified instruction arguments will correspond to a register in the hardware. However, due to the binding process of HLS, a single hardware register may correspond to data storage for multiple different LLVM-IR instructions, each with a unique execution trace. Thus, the trace data structure is organized to be aware of register binding, and keep multiple independent (potentially interleaved) traces for each register. For each trace, the values are stored in order (e.g. the first execution of Inst1 produces val_ex1, the second produces val_ex2, and so on).

```
// targeted load instruction
%tmp0 = load i64* %global_ptr0

// bitcast
%tmp1 = bitcast i64 %tmp0 to i32

// alignment with shift and extend
%tmp2 = lshr i32 %tmp1, 32
%tmp3 = zext i32 %tmp2 to i64

// call the function to record this value into trace
call void @recordtrace(i64 %tmp3)
```

Figure 6.3: Instrumentation Code

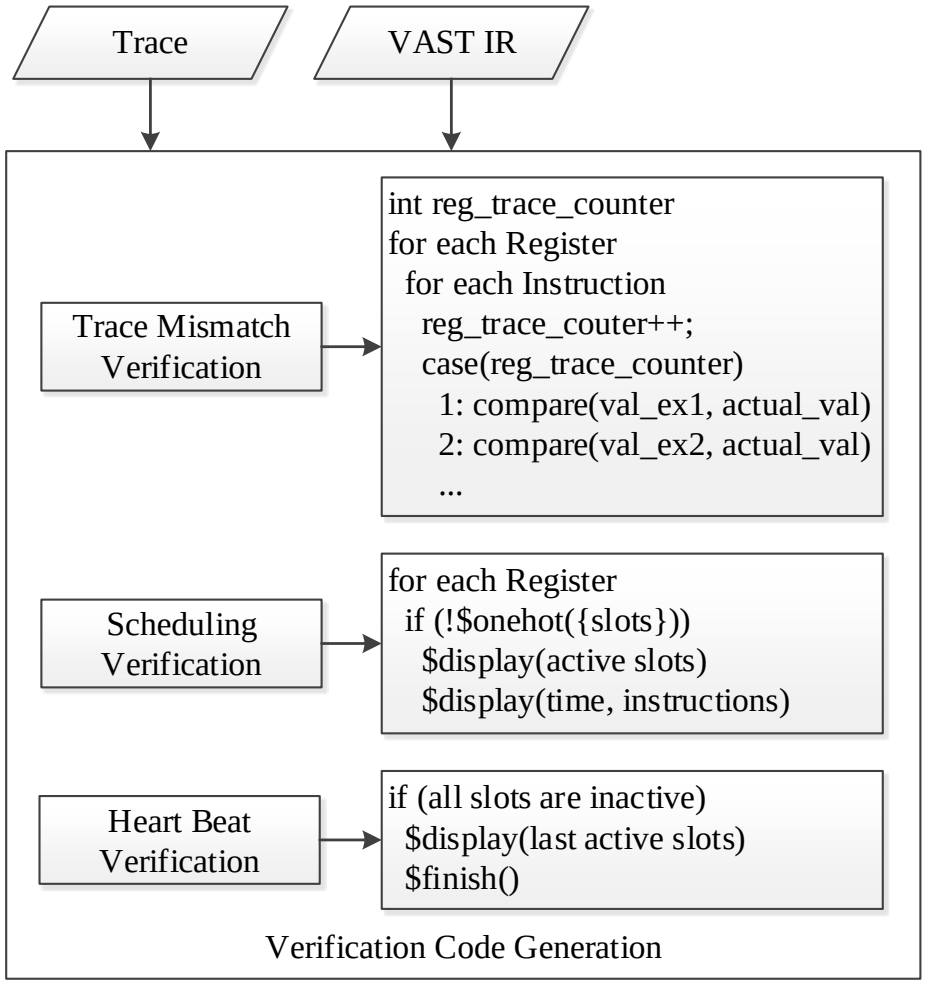

Figure 6.4: Verification Code Generation

6.2.2 Verification Code Generation

Given the trace data from software execution, we can now generate the verification code as part of the RTL code generation. As shown in Figure 6.4, for each register and LLVM-IR instruction, we generate a block of verification code. Each of these verification blocks uses a trace counter to keep track of the sequence of trace values for that LLVM-IR instruction. In addition to the instruction specific blocks, which verify the LLVM-IR invariants, we generate scheduling verification and heart beat verification code blocks to verify hardware-specific invariants as discussed in Section 6.1.

```
case (mem1p0wdata_2726930_trace_counter)
0: begin
  if ((32'hF0) != (w102w[31:0])) begin
    $display("%t:Trace mismatch!",$time());
    $display("Expected: 32'hF0,");
    $display("Actual: %x", (w102w[31:0]));
    $finish(1);
  end
end
1: begin
  if ((32'hF1) != (w102w[31:0])) begin
    $display("%t:Trace mismatch!",$time());
    $display("Expected: 32'hF1,");
    $display("Actual: %x", (w102w[31:0]));
    $finish(1);
  end
end
```

Figure 6.5: Trace Mismatch Verification

```
if (!$onehot0({(Slot22_main_r),
               (Slot23_main_r)}))
begin
  $display("At time %t, register
  mem6p0addr has more than one active
  assignment: %b!", $time(),
  {(Slot22_main_r), (Slot23_main_r)});
  if (Slot22_main_r) begin
   $display("%t:Condition:(Slot22_main_r),
    Src: (w82w[11:0]), current slot: 22,
    for.body, main", $time());
    /*%tmp48 = load i32* %tmp47, align 4*/
  end
  if (Slot23_main_r) begin
   $display("%t:Condition:(Slot23_main_r),
    Src: (w84w[11:0]), current slot: 23,
    for.body, main", $time());
    /*%tmp60 = load i32* %tmp59, align 4*/
  end
  $finish(1);
end
```

Figure 6.6: Scheduling Verification

For each instruction, we insert a trace mismatch verification block as in Figure 6.5. Using the trace counter, this code block tracks an expected value for each execution of

the instruction, and in the case of mismatch prints information about the mismatch. For brevity, it is not shown in Figure 6.5, but the generated code also includes the corresponding LLVM-IR instruction and its parent basic block to aid in bug diagnosis. In the case of a trace mismatch, we also classify mismatches into critical and non-critical faults. Critical faults indicate unrecoverable execution mismatches, and thus we immediately terminate simulation. Non-critical faults may be execution mismatches due to predication (e.g. due to global code motion): the instruction execution may be canceled and re-executed with updated arguments, producing correct functionality. If the fault is non-critical, we print warnings, but do not halt simulation. If the non-critical fault represents a real fault, the erroneous execution will propagate to a critical fault.

In addition to instruction-based verification blocks, we also verify that each register is only used by one instruction at a time. As discussed in Section 6.1, VAST HLS generates a one-hot FSM, and we know which state each instruction is scheduled to. Thus, as shown in Figure 6.6, we generate a verification block that ensures that each instruction using a particular hardware register is activated mutually exclusively by *slot22_main_r* or *slot23_main_r*. Failure in a one-hot verification block is always a critical fault. As in trace mismatch, the verification code also includes LLVM-IR instruction information to aid in bug diagnosis.

As shown in Figure 6.7, we also include a simple verification block as a watchdog timer to ensure that a simulation always concludes, whether successfully or unsuccessfully within a worst-case simulation latency, and prints out the active state which might be related to the cause that leads to an unexpected timeout, e.g. incorrect looping or branching.

Finally, we also generate a verification block to ensure that the FSM is always in a valid state as shown in Figure 6.8. This heart beat block simply ensures that at least one state is always active; if no state is active, the RTL module FSM has died unexpectedly.

In summary, we generate verification code blocks for all load, store, branch, switch and ϕ instructions and cover all LLVM-IR program invariants. When provided with

```
// watchdog timer
module watchdog_timer
  #(MAX_COUNT=`WORST_CASE_LATENCY) (
  input clk,
  input rstN);

  int counter=MAX_COUNT;

  always@(posedge clk) begin
    if(!rstN)
      counter <= MAX_COUNT;
    else
      counter <= counter - 1'b1;

    if(counter == 0)
      // print currently active state
      $finish(1);
  end

endmodule
```

Figure 6.7: Watchdog Timer

```
if (!(Slot1_main_r | Slot2_main_r |
      Slot3_main_r | Slot4_main_r |
      Slot5_main_r | Slot6_main_r |
      Slot7_main_r | Slot8_main_r ))
begin
  $display("At time %t, no heart beat.\n",
           $time());
  $finish(1);
end
```

Figure 6.8: Heart Beat Verification

multiple test vectors (and thus multiple trace data) for each benchmark, the verification framework will detect all bugs in the RTL implementation as the simulation will produce an execution mismatch in at least one load, store, branch, switch or ϕ instruction. Furthermore, because our verification code automatically prints diagnostic information and halts simulation at the first critical execution mismatch, we ensure low-latency detection of bugs and accurate pinpointing of the offending LLVM-IR instruction(s).

6.2.3 Application to Other HLS Tools

Although implemented in VAST HLS, this strategy could be implemented on other HLS tools. To add this verification code to generated RTL, we require:

1. An input trace for a set of test vectors with data for each critical instruction
2. A mapping between C-level or LLVM-IR level instructions and RTL-level variables
3. A mapping of binding information to determine which LLVM-IR instructions use the same RTL-level variable
4. The scheduling result for each instruction to know the expected state machine states for each assignment

With this information, we can generate all of the implementation specific verification code. We could also determine appropriate analogs for platform-specific verification code (e.g. in a platform with grey-coded FSM states, we could generate appropriate verification statements to ensure that shared objects aren't activated by more than one user at a time).

6.3 Results

6.3.1 Experimental Setup

We now study the feasibility and effectiveness of our proposed technique. First, we will evaluate the overhead of our verification technique in terms of HLS execution time as well as RTL simulation time. Then, we will introduce our random bug insertion framework, and evaluate bug detection rate and latency using randomly inserted bugs to evaluate the efficiency of the debug process. After that, we demonstrate debugging cases for the verification of our VAST HLS kernel. These bugs include a variety of potential problems that an HLS developer may encounter while implementing a new feature, but should not be considered a *complete* set of detectable bugs: any HLS

kernel bug that produces incorrect RTL output, which will trigger a mismatch in one of the invariants, will be detected by this technique.

6.3.2 Overhead of our Verification Technique

As a pre-synthesis verification technique for functional correctness, the overhead of our proposed technique mainly consists of two parts: overhead in HLS process time and overhead in simulation time. HLS overhead is due to additional time for trace generation, and verification code generation. Simulation overhead is due to additional complexity in simulating a larger RTL file that includes a potentially large amount of verification statements.

Our verification technique does not add area overhead to synthesized designs, as our code does not insert hardware instrumentation, top level ports, or trace gathering buffers. All of our verification code is generally non-synthesizable, so it is inserted in a SYNTHESIS OFF region in the RTL such that it is automatically ignored during logic synthesis.

For each benchmark from the CHStone suite, we measure the HLS core execution time with and without the verification code feature. Similarly, we measure the simulation time of HLS-generated RTL with and without added verification code. In Figure 6.9, we show the overhead of HLS core time and simulation time with respect to the version without verification code.

In terms of HLS core time, there is an average 25% overhead across all benchmarks, with a maximum overhead of $\tilde{5}0\%$ for jpeg, the benchmark with the largest execution trace. In all cases, the total HLS core time is less than one minute both with and without verification code. Although our method requires one HLS generation for each set of test vectors, this is still a small and acceptable overhead to generate more detailed debugging information.

In simulation time, the extra complexity of verification code can substantially increase total simulation time of **functionally correct** outputs. However, in practice, this method should be used to help identify the location of functional mismatch in functionally incorrect code; we do not suggest inserting verification code when not

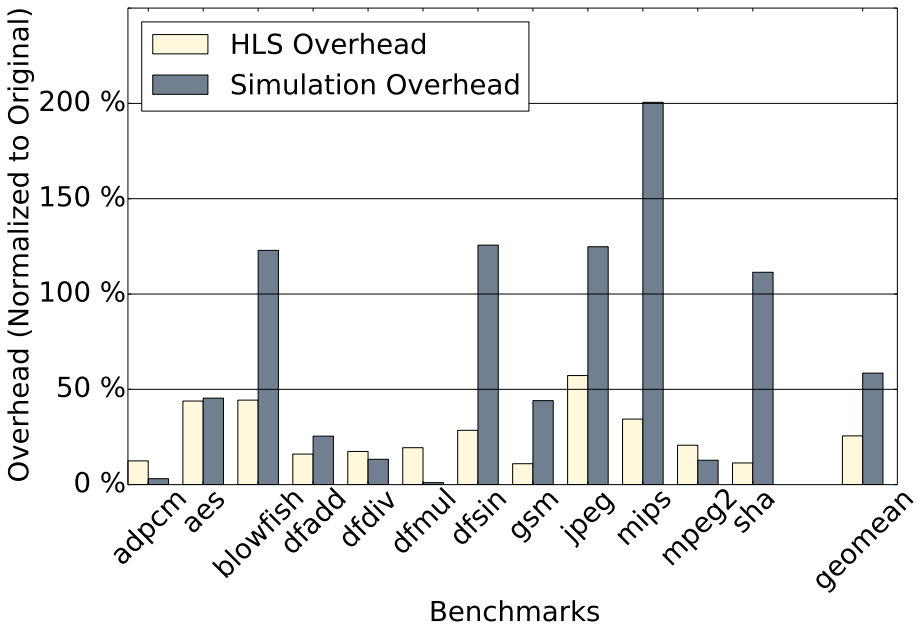

Figure 6.9: HLS and Simulation Overhead

performing debugging. The overhead ranges from almost no overhead (dfmul) to about 3× (mips). The benchmark complexity corresponds to increased complexity in memory access patterns, longer test vectors (which increases the number of values to verify), and a larger total number of verification code blocks.

Some benchmarks such as mips and jpeg have complex memory access patterns including multi-dimensional data structures, and use of sub-routines. Their control flow includes many conditional code blocks and complex loops that require ϕ node verification, and large number of load/store instructions which increase the total number of verification blocks. However, although the simulation overhead in correct RTL can be as much as 3×, the early exit conditions usually exit the simulation earlier, once a mismatch is found.

In total, the average simulation overhead for **functionally correct** simulation is 58%. However, as we will demonstrate in the next sub-section, early exit conditions eliminate that overhead, and actually accelerate the process: on average, the bug is detected and simulation halted faster than a full, functionally correct simulation, with an average speedup of 2×. This result demonstrates that, compared to traditional

debugging strategy where the debugger needs to wait for the completion and relies on the final result to investigate the cause of bugs, this debugging framework not only speeds up the simulation process, but also provides diagnostic information about the mismatch to further facilitate the debugging process. Therefore, these features accelerate the verification and debugging process with improved debugging efficiency, which also contribute to the overall productivity improvement.

6.3.3 Effectiveness Evaluation

Evaluation Infrastructure

In order to demonstrate the effectiveness of our proposed verification technique, we use our extended HLS framework with the applications from the CHStone benchmark suite [153]. We can generate software traces by executing the benchmarks and automatically insert the corresponding verification code in the HLS-generated RTL, but the verification technique cannot be evaluated if there are no bugs in the generated RTL.

In view of this, we develop an automated testing script that inserts bugs into the HLS-generated Verilog RTL code. In our automated bug-insertion flow, we randomly select an RTL operator, then randomly select one line of code within the RTL containing that operator, then replace one instance of that operator with another operator. This method ensures that the result of all of our random modifications remains *syntatically correct*, although functionally incorrect.

Though we only insert bugs by operator replacement, this strategy still captures multiple scenarios in which an RTL bug can occur: (i) if an operator is changed in an arithmetic or logical evaluation, it is equivalent to changing the operands in the statement as both would produce an eventual bug (unless the test vectors do not activate the bug, as discussed in Section 6.1) , and (ii) if an operator is changed in a statement where state transitions are evaluated, it will correspond to altering the state transitions. Thus, with the operator replacement strategy, we are able to modify both the data-path and control-path portions of the RTL code.

In addition, our testing script also inserts additional code in the generated RTL

to compute detection latencies. We create a counter and store the values for both the *original* operator and *replacement* operator. After a detection of HW/SW discrepancy, the counter increments the latency in clock cycles until the bug is detected and the simulation completes. It is important to emphasize that this counter is only for the purposes of our evaluation framework; with this addition code, we can gather statistical information about bug detection latency, but in practice we obviously cannot insert this code before knowing where the bug is.

Our testing script gathers detailed information on the detection latency, and total simulation latency. The verification code inserted by our framework also prints out detailed information about the LLVM-IR instruction(s) involved in the mismatch, expected and actual values, and simulation cycle of the mismatch. In our testing script, the bug insertion and detection is automated, but in practice the diagnostic information printed by our framework would aid the user in quickly and efficiently identifying associated hardware structures such as registers, multiplexers, FSM states, and connected intermediate variables. We will demonstrate how this diagnostic information can be used in more detail in Section 6.3.5.

Effectiveness Results

For each CHStone benchmark, we execute 1000 simulations with a single, randomly inserted bug. A bug is considered detected if the simulation ended due to any of our verification code blocks, and undetected if the simulation ends without any detected error. Table 6.1 shows the number of bugs inserted, detection rate, average simulation time, and average simulation speedup with respect to the simulation time of non-faulty RTL without verification code.

In total 7 of 12 benchmarks achieve 100% detection accuracy, as their test vectors are sufficient to cover all corner cases in the HLS-generated RTL. In the remaining benchmarks we detect 83% or greater in all cases, with an average across CHStone of 94.8%. In the undetected cases, they could be detected through additional test vectors. In several cases, we verified that an originally undetected bug simply is not activated by the existing test vectors. By using alternate test inputs, re-generating the verification code, and re-inserting the identical bug, we can identify additional

Table 6.1: CHStone Bug Results

CHStone Benchmarks	Number of Iterations	Detection Rate	Average Sim Time (s)	Sim Time Speedup Over Full Simulation
adpcm	1000	88.5%	0.434	1.53×
aes	1000	86.3%	1.292	3.88×
blowfish	1000	89.5%	90.725	4.15×
dfadd	1000	100%	0.748	1.13×
dfdiv	1000	100%	0.432	1.86×
dfmul	1000	100%	0.573	1.17×
dfsin	1000	100%	19.026	1.53×
gsm	1000	83.3%	4.009	1.23×
jpeg	1000	90.5%	731.687	2.18×
mips	1000	100%	0.808	3.97×
mpeg2	1000	100%	1.530	2.75×
sha	1000	100%	125.396	1.64×
overall	12000	94.8%	4.563	2.01×

source bugs.

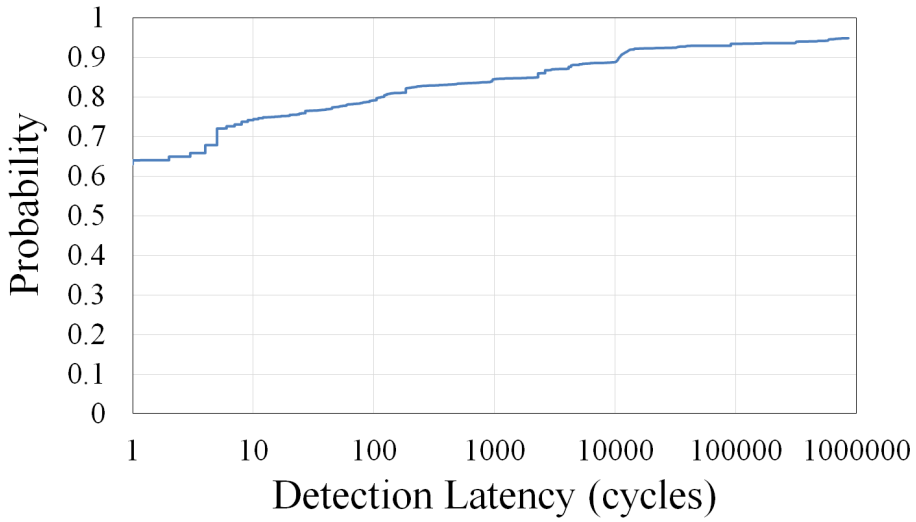

Figure 6.10: Cumulative Distribution Function of Bug Detection Latencies

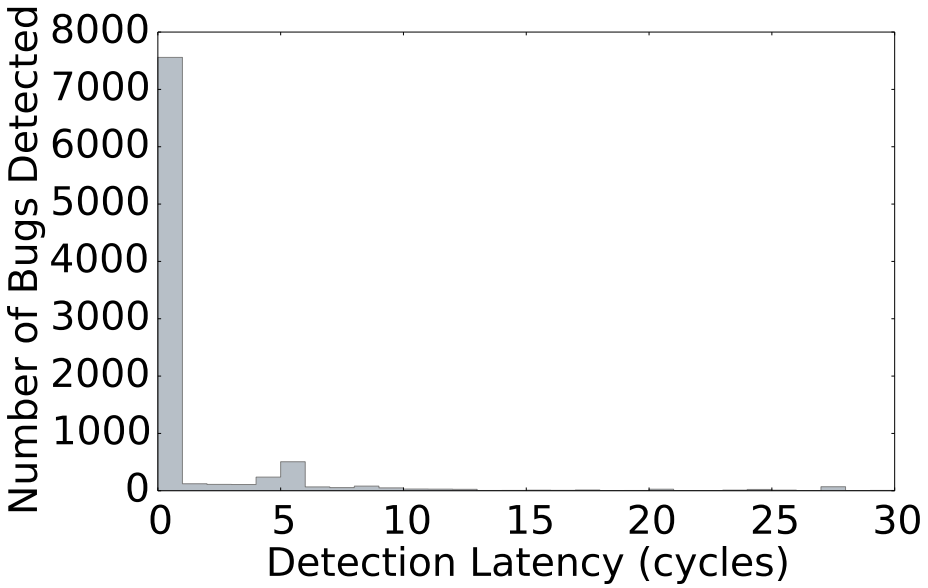

Figure 6.11: Histogram of Bugs Detected (0 to 30 cycles)

On average, the verification code identifies a bug within a few cycles of creating the functional mismatch; 78% of bugs are found within 10 cycles of the first instance of functional mismatch. Figure 6.10 shows the cumulative distribution function of bug detection latencies. As we can see, the vast majority of bugs are found quickly. Among longer detection latency bugs, several include incorrect flag computation at the beginning of a simulation that does not impact correctness until the function exit conditions, and control-flow bugs that prevent the core from ever finishing correctly (and thus are found via the watchdog timer).

We look in more detail for bugs detected in low latency; Figure 6.11 shows a histogram of bug detection latency for bugs detected between 0 and 30 cycles. In this set, the vast majority of bugs are detected with 0 cycle latency: over 15× more bugs than at any other latency. There are some relatively important groups of bugs detected in the range of 1-10 cycles, but beyond 11 cycles there are comparatively fewer bugs detected. Although there are some important bugs detected with higher latency, this demonstrates that the majority of detected bugs are found with low-latency.

In total, these random bug insertion tests demonstrate that our proposed detection

scheme both effectively finds bugs whenever the test vectors activate the bug, we detect the functional mismatch of the majority of bugs in only a few cycles, and average simulation of latency is sped up relative to simulation of functionally correct RTL without verification code.

6.3.4 Cases of HLS Kernel Bugs

In addition to the evaluation conducted above, we demonstrate our trace-based verification framework is particularly useful to assist in debugging and diagnosis of HLS kernel bugs through several debugging cases. Using the (source-level verified correct) CHStone suite of benchmarks, we use the verification framework to detect 9 representative HLS kernel bugs. These examples are representative of a variety of HLS bugs that may be introduced during development of new features.

It is important to note that the HLS development and verification process must test the HLS kernel on a wide number of benchmarks. For all of these bugs, the produced RTL is incorrect in only a subset of CHStone benchmarks; often for only one specific combination of HLS optimization options. The modularity of VAST HLS to enable and disable individual optimizations helped to narrow the faulty test cases, but the bug itself is often a corner-case combination of the exact combination of program representation and the results of multiple interrelated HLS optimizations.

Table 6.2 shows the bug detection results for the 9 representative HLS core bugs. The bug detection latency for 7 of the 9 bugs is zero: at the instant the bug produces incorrect results, the verification code detects incorrect execution and immediately halts simulation. The only non-zero detection latency is a bug in the translation of a combinational function; because it takes several cycles for an incorrect computation to propagate to a load or store, the detection latency is a few cycles. Together these bugs demonstrate automated coverage of potential errors in any portion of the HLS core: bugs in control FSM generation, pipelining, translation of combinational operations (e.g. due to bit level optimization), memory initialization, scheduling, and binding (specifically, the port assignment problem) are examples that represent the entire HLS flow.

Table 6.2: HLS-Core Bug Results

Bug Description	Related LLVM IR	Detection Latency	Detection Method
Incorrect state transition	Control-state load	0	Trace mismatch
Missing register for MUX pipelining	Pipeline-register store	0	Trace mismatch
Missing register for datapath pipelining	Pipeline-register load	0	Trace mismatch
Incorrect computation translation	Data-dependent load	3	Trace mismatch
Incorrect computation translation	Data-dependent store	3	Trace mismatch
Incorrect array initialization	Initialization load	0	Trace mismatch
Incorrect handling of control flow for PHI	PHI-dependent load	0	Trace mismatch
Conflicting port assignment	Memory-port loads	0	Scheduling verification
Control state transition broken	Control-state PHI	0	Heart beat detection

Together, the effectiveness evaluation and our example verification cases demonstrate that:

- The proposed detection schemes effectively covers HLS kernel bugs that cause incorrect behavior in at least one of our invariants with high coverage.

- Trace-based verification can improve detection latency significantly.

- Trace-based verification requires only mild overhead in HLS execution time and simulation time, but can achieve a speedup when bug is detected.

- Trace-based verification can provide detailed information facilitating debugging and diagnosis of the HLS kernel bugs.

6.3.5 User Debug Process

We have now demonstrated that our verification process is effective to detect bugs with low detection latency and denoting the earliest execution mismatch in RTL including identification of LLVM-IR instruction(s) and hardware structures in RTL. However, simply identifying this functional mismatch accelerates only the bug detection portion of the verification process. We now demonstrate the debug process with several examples in order to show how the diagnostic information from our verification code can help a user identify the cause of functional mismatch and further accelerate the debug process for productivity improvement.

In the first experiment, we retained the HLS-generated *correct* RTL of mips benchmark but modified the HLS-generated memory initialization file such that the bug results from incorrect static initialization of data for internal memories. Our verification code correctly detected a mismatch and printed diagnostic information. With the information that a load value was incorrect, the user can verify the data collected in this load instruction and then examine the initialization data file and determine that the file was missing a single value, and thus trace the cause of that bug. It is important to note that, although running the source code in software can show us in the final result that some unknown value in the whole application is incorrect and results in the incorrect final result, our method provides more detailed diagnostic information about which instruction is responsible for the incorrectness.

In the second experiment, the bug is in the multiplexer pipelining logic. Mux pipelining improves the achievable frequency by replacing a large mux (e.g. to share memory ports) with a pipelined mux tree. We introduce a bug that incorrectly does not insert a pipeline register on one of the paths. We rerun all of the CHStone benchmarks using the modified HLS framework and generate RTL. In many benchmarks, this corner case was either not activated, or the register was missing, but without affecting functional correctness (e.g. if the incorrect path is never used within a few cycles of conflicting paths, it may never produce incorrect results). However, the jpeg benchmark failed simulation and detected the mismatch. With the diagnostic information from the mismatch, we know that a store instruction stores an incorrect value. After tracing to the functional unit we determine that the result is correctly

produced, but somehow the value does not propagate from storage to the memory port correctly. Thus, we identify the pipelined mux structure, and eventually that a register is missing. The diagnostic information printed by our verification code is shown in Figure 6.12.

```
Time: 8955415:
following instruction:
store i32 %tmp130, i32* %scevgep47.i.i, align 8
in RTL line: 27332
Trace mismatch!
Expected: ((32'hE3)), actual: 000000e4!
```

Figure 6.12: Diagnosis in Mux Pipelining

Although some bugs can be detected simply, others require more detailed analysis. We create a bug in the port assignment code segment that distributes memory operations between two ports. In this case, the port assignment code is always correct for single basic blocks, but is only incorrect when memory instructions from multiple basic blocks execute at the same time (due to global code motion [143]), and they are assigned to the same port. This combination of cases only occurs in the aes benchmark with a certain combination of optimization options and scheduler settings when using global code motion. The diagnostic information printed by our verification code is shown in Figure 6.13.

```
At Time: 387433, register mem6p0addr
has more than one active assignment: %b!
slot w80w[11:0] %tmp48 = load i32* %scevgep29, align 4,
Parent BB: .thread
Parent Slot: .thread <Launch>
slot w202w[11:0] %tmp60 = load i32* %scevgep12, align4,
Parent BB: .preheader3
Parent Slot: .preheader3 <Launch>
```

Figure 6.13: Diagnosis in Port Conflict

In this case, the debug information tells us that there are two LLVM-IR instructions sharing a single memory that are incorrectly attempting to use the same memory in the same cycle. The error message initially tells the user that these two loads are

receiving unexpected values, but also specifies that two loads that should not execute at the same time are both active. From there, the user can determine that the two instructions come from different basic blocks and infer that there is a relation to global code motion or incorrect control transfer. After eliminating both of these as direct causes of the bug, we can identify that although the memory port assignment algorithm is aware of memory parallelism, it does not correctly evaluate parallelism between parallelly executing basic blocks. In this case, the diagnostic information is particularly valuable; this bug was only activated in a single application for a unique combination of HLS optimizations.

6.4 Conclusions

In this chapter, we developed an HLS-based verification framework that can facilitate the debugging process of HLS kernels by identifying the earliest execution mismatch in any faulty RTL implementation with low detection latency. We implement this framework using software-based trace generation and automated generation of verification code blocks for every critical instruction (all load, store, branch, switch and ϕ instructions). We also developed an automated framework to insert bugs in the HLS-generated RTL and demonstrate that our auto-generated RTL verification code detects 94.8% of the 12000 inserted bugs, and 78% of the detected bugs within 10 cycles. Additionally, we demonstrate how the diagnostic information generated by our verification framework can be used to identify the cause of source-level or HLS bugs. This facilities significantly improve the efficiency of debugging process of HLS tools, leading to faster delivery of HLS optimizations which contribute to the overall productivity.

Chapter 7

Conclusions and Future Work

In this chapter, we conclude by summarizing how our proposed solutions address the productivity challenges in reusability and verification, and discussing future work in productivity improvement for hardware design flows, and particularly for further productivity improvement of HLS-based design flows.

7.1 Conclusions

As the complexity of applications and thus hardware designs continues to grow, productivity of hardware design flows has become a major challenge. High level synthesis is a promising methodology for improving productivity by raising the design entry abstraction level from RTL to high level language (behavioral) level descriptions. However, despite the achievements in productivity improvement, there are remaining productivity bottlenecks. Particularly, integration with reusable IPs and design verification have become particular productivity bottlenecks in HLS-based design flows.

In Chapter 4, we presented a generalized behavioral-level IP integration framework. By using flexible user-specifiable mappings between instructions or functions and instantiated IP blocks, we support a wide variety of IP blocks as both internal instantiations and as system-level integrated IP blocks. This framework also features support for fixed- and variable-latency IPs, and non-synthesizable IPs. With all of

these features, we presented several case studies to demonstrate that our behavioral-level IP integration strategy can achieve significant productivity improvement by easily specifying various IP sources for more efficient evaluation of different IP implementations and design space exploration. In addition, the compatibility between IP integration and HLS optimizations leads to quick delivery of optimized QoR for reduced design iterations, thus also contributes to the overall productivity. Generalized IP support leads to a wide variety of development flows for HLS including incremental HLS, support for non-synthesizable IP blocks for debug and validation, design partitioning to remove expensive or undesirable sub-functions, as well as more complex flows such as reliability-oriented HLS where every functional unit is replaced by customized components for increased reliability. Furthermore, behavioral-level IP integration resolves challenges in efficiently integrating IPs for exploiting parallelism and partitioning of applications, thus saving considerable design effort and refinement iterations compared to manual exploration in system-level strategy. In summary, all of these features and supports provided in this solution contribute to significant productivity improvement.

Next, we address time-consuming source-level debugging of HLS-produced designs in Chapter 5. We presented AutoSLIDE, an automated verification framework for HLS that is able to pinpoint the exact origin of design bugs in the source code. AutoSLIDE instruments applications, and uses software-generated traces to insert verification code into HLS-produced RTL. On an execution mismatch, the framework backtraces the datapath to automatically instrument intermediate nodes for a more precise diagnosis of the instruction(s) in C/C++ causing the execution mismatch. We demonstrate the effectiveness of our framework by automatically detecting and precisely localizing bugs from former versions of the CHStone benchmark suite. This automated process considerably eliminates the manual effort of precisely localizing bug sources related to HW/SW execution mismatch, substantially enhancing the debugging efficiency and improving productivity. Furthermore, we demonstrate that our solution features low HLS run-time and simulation time overhead, as well as reductions in trace size and trace gathering time compared to exhaustive approaches, thus achieving improved efficiency in verification productivity through the savings in

HLS run-time, simulation time, and the effort in trace collection and comparison.

Finally, in Chapter 6, we further address productivity through improved debugging of HLS tools and thus indirectly improved productivity through reduced user effort and fewer design iterations to meet QoR objectives. We developed an HLS-based verification framework that can facilitate the debugging process of the development of HLS optimizations. It identifies the earliest execution mismatch between the software-generated traces and the simulation with faulty RTL implementation with low detection latency. This framework features automated software-based trace generation and automated generation of verification code blocks for every critical instructions to guarantee the functional correctness of HLS-generated RTL output. We also developed an automated framework to insert bugs in the HLS-generated RTL and demonstrate that our auto-generated RTL verification code detects 94.8% of the 12000 inserted bugs, and 78% of the detected bugs within 10 cycles. We demonstrate how the diagnostic information generated by our verification framework can be used to identify the cause of source-level or HLS bugs. This facilities significantly improve the efficiency of debugging process of HLS tools, leading to faster delivery of HLS optimizations which contribute to the overall productivity.

7.2 Future Work

This section discusses the remaining problems that still require non-trivial manual design effort in the HLS-based design methodology.

7.2.1 Reusability & IP Integration

Behavioral-level IP integration addresses the efficiency of integrating IPs for improved productivity, but there are several remaining productivity challenges in IP integration. Given a complex HLL application and an IP library, the challenge of determining whether a given IP is functionally identical to an RTL IP from the library is

extremely challenging. If the IP is not generated by prior HLS of the HLL description, it is yet more challenging. Even comparing two control data flow graphs (CDFGs) to determine isomorphism is NP-hard, yet verification between RTL and HLL is even more challenging – the different abstraction level may lead to graphs with significantly different numbers of nodes and communications between nodes for two functionally identical graphs. Therefore, the challenge in determining whether the HLL description and the mapped RTL IP implementation are functionally identical requires considerable human effort; current techniques in both formal verification and simulation-based verification are insufficient to automate the process of determining whether HLL function descriptions are functionally identical to RTL IPs, and thus the challenge of determining which IPs should be used is a productivity bottleneck.

In addition, decision making of how to partition the overall application into portions which are implemented separately by various IPs requires considerable exploration effort, thus also limiting the productivity. For example, an application can be partitioned either into larger portions which are implemented in more complex IPs, or into smaller portions which are implemented in less complex IPs. Users have to explore various partitioning schemes to determine the most appropriate way for mapping the portions into separate IPs. There is a need to reduce the human effort in exploring different partitioning schemes to quickly choose the appropriate one that achieves good QoR and improved productivity.

There are also other factors in design space exploration that need to be addressed. First, there may be many implementations of an IP that trade-off between performance and area usage. Design space exploration must effectively select which IP configurations best meet design objectives, and also whether the same IP configuration should be used for all instantiations or different configurations should be used for different instantiations. Finally, design space exploration must also explore how many instantiations should be used and the binding of individual function (IP) calls to IP instantiations. All these design space exploration factors remain bottlenecks to achieving further improved productivity due to the significant exploration effort required.

In addition, although HLS scheduling can effectively parallelize fixed-latency IPs

and other operations, it remains challenging to overlap variable-latency IP executions with other operations for improved parallelism. This requires a mechanism for parallel state machine control and automatically generated synchronization methods to leverage parallelism. Furthermore, in order to explicitly represent parallelism and data-dependencies in high level source, IP integration may need to support non-blocking function calls and other techniques for representing coarse-grained parallelism.

In cases where multiple IPs have to access multiple items in the same memory, coarse-grained dependence analysis of these IPs requires that the following IP can only start after the previous one finishes. Thus there is a need for fine-grained analysis of the IPs so that the following IP can be partially overlapped to the previous one even before the previous one finishes for enhanced parallelism. However, HLS kernel schedules IP-implemented operations without any knowledge about the implementation details inside of the RTL IPs. Thus, it is still challenging to exploit the memory-level parallelism when multiple IPs have the access to the same memory, thus also resulting in sub-optimal performance which in turn lowers productivity by requiring more design iterations for performance refinement.

7.2.2 Verification

We presented AutoSLIDE for source-level debugging in Chapter 5, and trace-based verification of HLS tools in Chapter 6, to directly and indirectly improve the debugging productivity. However, as the complexity of design increasingly grows, scalability stands out as a bottleneck to productivity improvement. First, the verification process requires a large set of extra logic for instrumentation. This extra logic significantly slows down the simulation speed. Minimizing instrumentation, reducing simulation range, or reducing the set of test vectors to improve scalability while still meeting test objectives is critical for verification productivity.

In addition, despite that in order to improve verification productivity, our framework has been evolving to enabling automated extraction of test vector so that the human developer do not need to manually create the test vector for verification, test vector selection is a critical challenge. Selecting the test vectors in order to maximize

code coverage is important in general testing, but in HLS-produced design there are additional challenges in ensuring that test vectors cover sufficient corner cases in both high-level source as well as the generated RTL. Furthermore, due to the scalability issues, it is critical that the test vector selection maximizes coverage while minimizing the total set of tests in order keep verification time minimal.

Together, this represents a critical need for productivity of verification; emulation-based approaches improve verification run-time, but create challenges in effectively tracking large numbers of signals or automating signal comparison and bug-tracing. Our approach automates signal selection, tracing, and bug localization but relies on simulation-based approaches that may have long run-time. Effective verification must help to improve the verification flow in general by improving test vector selection, accelerating verification run-time, automating signal selection, and accelerating bug-tracing and identification.

Bibliography

[1] Altera Inc. *Altera Development Kit, Stratix V Edition User Guide*. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_svgs_dsp_dev_kit.pdf.

[2] The International Technology Roadmap for Semiconductors (ITRS), System Drivers, 2009, http://www.itrs.net/.

[3] M. Meredith and S. Svoboda. The Next IC Design Methodology Transition Is Long Overdue, Open SystemC Initiative. 2010.

[4] Altera Inc. *Nios II Processor Reference*. http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf.

[5] Xilinx Inc. *MicroBlaze Processor Reference Guide*. http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf.

[6] ARM Holdings. *ARM Cortex-A Series*. https://static.docs.arm.com/den0024/a/DEN0024.pdf.

[7] Synopsys, Inc. *Virtualizer Datasheet*. http://www.synopsys.com/Solutions/IndustrySegmentSolutions/MilAero/Documents/virtualizer_ds.pdf.

[8] Cadence Design Systems, Inc. *Cadence Virtual System Platform*. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/Archive/virtual_system_platform_ds.pdf.

[9] Y. Park, H. Park, and S. Mahlke. Cgra express: Accelerating execution using dynamic operation fusion. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 271–280, New York, NY, USA, 2009. ACM.

[10] L. Wan, C. Dong, and D. Chen. A coarse-grained reconfigurable architecture with compilation for high performance. *Int. J. Reconfig. Comput.*, 2012:3:3–3:3, January 2012.

[11] B. De Sutter, P. Raghavan, and A. Lambrechts. *Coarse-Grained Reconfigurable Array Architectures*, pages 449–484. Springer US, Boston, MA, 2010.

[12] B. De Sutter, P. Coene, T. Vander Aa, and B. Mei. Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '08, pages 151–160, New York, NY, USA, 2008. ACM.

[13] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck. Spr: An architecture-adaptive cgra mapping tool. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 191–200, New York, NY, USA, 2009. ACM.

[14] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins. Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: A case study. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '04, pages 21224–, Washington, DC, USA, 2004. IEEE Computer Society.

[15] N. W. Bergmann, S. K. Shukla, and J. Becker. Quku: A dual-layer reconfigurable architecture. *ACM Trans. Embed. Comput. Syst.*, 12(1s):63:1–63:26, March 2013.

[16] G. Stitt and J. Coole. Intermediate fabrics: Virtual architectures for near-instant fpga compilation. *IEEE Embedded Systems Letters*, 3(3):81–84, Sept 2011.

[17] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, and D. L. Maskell. Virtualized execution and management of hardware tasks on a hybrid arm-fpga platform. *Journal of Signal Processing Systems*, 77(1):61–76, 2014.

[18] D. Capalija and T. S. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–8, Sept 2013.

[19] A. K. Jain, S. A. Fahmy, and D. L. Maskell. Efficient overlay architecture based on dsp blocks. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 25–28, May 2015.

[20] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy. Deco: A dsp block based fpga accelerator overlay with low overhead interconnect. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24rd Annual International Symposium on*, May 2016.

[21] Xilinx Inc. *Unleash the Unparalleled Power and Flexibility of Zynq UltraScale+ MPSoCs*. `http://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html`.

[22] N. Kapre and J. Gray. Hoplite: Building austere overlay nocs for fpgas. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2015.

[23] J. Gray. Grvi phalanx: A massively parallel risc-v fpga accelerator accelerator. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24rd Annual International Symposium on*, May 2016.

[24] N. Kapre. Marathon: Statically-scheduled conflict-free routing on fpga overlay nocs. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24rd Annual International Symposium on*, May 2016.

[25] *Clang/LLVM Maturity Report*, Moltkestr. 30, 76133 Karlsruhe - Germany, June 2010. *See* `http://www.iwi.hs-karlsruhe.de`.

[26] C. Lattner and V. 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–, Washington, DC, USA, 2004. IEEE Computer Society.

[27] Xilinx Inc. *Vivado High-Level Synthesis*. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/hls/index.htm`.

[28] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design and Test of Computers*, 26(4):8–17, 2009.

[29] Semico Research Corp. *System(s)-on-a-Chip: Changes in SoC Design Methodology II*, 2015.

[30] Mentor Graphics, Inc. *ModelSim - Leading Simulation and Debugging*. `http://www.mentor.com/products/fpga/model`.

[31] F. Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[32] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. Autopilot: A platform-based esl synthesis system. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis*, pages 99–112. Springer Netherlands, 2008.

[33] Altera Inc. *Quartus II Subscription Edition Software*. `http://www.altera.com/products/software/quartus-ii/subscription-edition/qts-se-index.html`.

[34] Xilinx Inc. *Vivado Design Suite*. `http://www.xilinx.com/products/design-tools/vivado/`.

[35] H. D. Foster. Trends in functional verification: A 2014 industry study. In *DAC*, pages 48:1–48:6, 2015.

[36] A. G. Schmidt and R. Sass. Improving fpga design and evaluation productivity with a hardware performance monitoring infrastructure. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 422–427, Nov 2011.

[37] B. Holland, A. D. George, H. Lam, and M. C. Smith. An analytical model for multilevel performance prediction of multi-fpga systems. *ACM Trans. Reconfigurable Technol. Syst.*, 4(3):27:1–27:28, August 2011.

[38] J. Agron. Domain-specific language for hw/sw co-design for fpgas. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, DSL '09, pages 262–284, Berlin, Heidelberg, 2009. Springer-Verlag.

[39] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo. Peace: A hardware-software codesign environment for multimedia embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):24:1–24:25, May 2008.

[40] C. Sullivan, A. Wilson, and S. Chappell. Using c based logic synthesis to bridge the productivity gap. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 349–354, Piscataway, NJ, USA, 2004. IEEE Press.

[41] N. Bombieri, H.-Y. Liu, F. Fummi, and L. Carloni. A method to abstract rtl ip blocks into c++ code and enable high-level synthesis. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–9, May 2013.

[42] H. Nikolov, T. Stefanov, and E. F. Deprettere. Automated integration of dedicated hardwired ip cores in heterogeneous mpsocs designed with espam. *EURASIP J. Emb. Sys.*, 2008, 2008.

[43] J. Monson and B. Hutchings. New approaches for in-system debug of behaviorally-synthesized fpga circuits. In *FPL*, pages 1–6, 2014.

[44] J. Goeders and S. Wilton. Effective fpga debug for high-level synthesis generated circuits. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[45] J. S. Monson and B. L. Hutchings. Using source-level transformations to improve high-level synthesis debug and validation on fpgas. In *FPGA*, pages 5–8, 2015.

[46] G. Jeffrey and W. Steven. Using dynamic signal-tracing to debug compiler-optimized hls circuits on fpgas. In *International Symposium on Field-Programmable Custom Computing Machines*, 2015.

[47] Y. Iskander, C. Patterson, and S. Craven. High-level abstractions and modular debugging for fpga design validation. *ACM TRETS*, 7(1):2:1–2:22, 2014.

[48] H. D. Foster. Why the design productivity gap never happened. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 581–584, Nov 2013.

[49] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, and A. Ivanov. System-on-chip: Reuse and integration. *Proceedings of the IEEE*, 94(6):1050–1069, June 2006.

[50] N. Calagar, S. D. Brown, and J. H. Anderson. Source-level debugging for FPGA high-level synthesis. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–8, 2014.

[51] K. Hemmert, J. Tripp, B. Hutchings, and P. Jackson. Source level debugger for the sea cucumber synthesizing compiler. In *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pages 228–237, April 2003.

[52] 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 *ISCAS*, pages 165–168, 2014.

[53] J.-S. Yang and N. Touba. Expanding trace buffer observation window for in-system silicon debug through selective capture. In *VTS*, pages 345–351, 2008.

[54] E. Hung and S. Wilton. Speculative debug insertion for fpgas. In *FPL*, pages 524–531, 2011.

[55] P. Graham, B. Nelson, and B. Hutchings. Instrumenting bitstreams for debugging fpga circuits. In *FCCM*, pages 41–50, 2001.

[56] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[57] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh. From opencl to high-performance hardware on fpgas. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534, 2012.

[58] Cadence, Inc. *Stratus High-Level Synthesis*, 2015. `http://www.cadence.com/products/sd/stratus/pages/default.aspx`.

[59] Calypto. *Catapult C Synthesis*. `http://www.calypto.com/catapult_c_synthesis.php`.

[60] L. Yang, M. Ikram, S. Gurumani, S. Fahmy, D. Chen, and K. Rupnow. Jit trace-based verification for high-level synthesis. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 228–231, Dec 2015.

[61] A. Mathur, M. Fujita, E. Clarke, and P. Urard. Functional equivalence verification tools in high-level synthesis flows. *Design Test of Computers, IEEE*, 26(4):88–95, July 2009.

[62] X. Feng and A. Hu. Early outpoint insertion for high-level software vs. rtl formal combinational equivalence verification. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 1063–1068, 2006.

[63] M. Fujita. Equivalence checking between behavioral and rtl descriptions with virtual controllers and datapaths. *ACM Trans. Des. Autom. Electron. Syst.*, 10(4):610–626, October 2005.

[64] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Design Automation Conference, 2003. Proceedings*, pages 368–371, June 2003.

[65] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[66] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formal verification of ssa-based optimizations for llvm. In *ACM SIGPLAN Notices*, volume 48, pages 175–186. ACM, 2013.

[67] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert memory model. In A. W. Appel, editor, *Program Logics for Certified Compilers*. April 2014.

[68] P. Castéran and Y. Bertot. Interactive theorem proving and program development. coq'art: The calculus of inductive constructions. 2004.

[69] L. Liu. Automated Debugging Framework for High-level Synthesis. Master's thesis, Electrical and Computer Engineering, 2013.

[70] K. A. Campbell, D. Lin, S. Mitra, and D. Chen. Hybrid quick error detection (h-qed): Accelerator validation and debug using high-level synthesis principles. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 53:1–53:6, New York, NY, USA, 2015. ACM.

[71] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. A. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[72] D. Chen, J. Cong, and P. Pan. Fpga design automation: A survey. *Found. Trends Electron. Des. Autom.*, 1(3):139–169, January 2006.

[73] P. Coussy, A. Baganne, and E. Martin. A design methodology for ip integration. In *ISCAS*, volume 4, pages 711–714, 2002.

[74] D. Verkest, K. Van Rompaey, I. Bolsens, and H. De Man. Coware—a design environment for heterogeneous hardware/software systems. In *Readings in Hardware/Software Co-design*, pages 412–426, 2002.

[75] F. Schirrmeister and A. Sangiovanni-Vincentelli. Virtual component co-design-applying function architecture co-design to automotive applications. In *IVEC*, pages 221–226, 2001.

[76] Altera Inc. *Qsys System Design Tutorial*. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/tt/tt_qsys_intro.pdf`.

[77] Altera Inc. *SOPC Builder*. `http://www.altera.com/literature/ug/ug_sopc_builder.pdf`.

[78] Xilinx Inc. *Vivado IP Integrator*. `http://www.xilinx.com/publications/prod_mktg/vivado\/Vivado_IP_Integrator_Backgrounder.pdf`.

[79] Altera Inc. *Introduction to the Avalon Interface Specifications*. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf`.

[80] Xilinx Inc. *AXI Interconnect*. `http://www.xilinx.com/products/intellectual-property/axi_interconnect.htm`.

[81] Synopsys, Inc. *Synphony High-Level Synthesis Solution*, 2012. `http://www.synopsys.com/`.

[82] Cadence, Inc. *C-to-Silicon Compiler*, 2012. `http://www.cadence.com/`.

[83] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. 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.

[84] Altera Inc. *Altera Floating Point Megafunctions*. `http://www.altera.com/products/ip/dsp/arithmetic/m-alt-float-point.html`.

[85] Xilinx Inc. *LogiCORE IP Floating-Point Operator*. `http://www.xilinx.com/products/intellectual-property/FLOATING_PT.htm`.

[86] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):542–555, March 2008.

[87] GNU Project. *GNU Compiler Collection*. `http://gcc.gnu.org/`.

[88] S. Sinha and T. Srikanthan. Ip-enabled c/c++ based high level synthesis: A step towards better designer productivity and design performance. *International Journal of Reconfigurable Computing*, 2014, 2014.

[89] Xilinx Inc. *LogiCORE IP Linear Algebra Toolkit v2.0 Product Guide (AXI)*. `http://www.xilinx.com/support/documentation/ip_documentation/linear_algebra_toolkit/v2_0/pg131-linear-algebra-toolkit.pdf`.

[90] Desmond Higham and John Burkardt. *BLACK_SCHOLES: Simple Approaches to the Black-Scholes Equation*. `http://people.sc.fsu.edu/~jburkardt/c_src/black_scholes`.

[91] D. Eastlake, 3rd and P. Jones. Us secure hash algorithm 1 (sha1), 2001.

[92] Xilinx Inc. *SDSoC Development Environment*. `http://www.xilinx.com/products/design-tools/software-zone/sdsoc.html`.

[93] Xilinx Inc. *LogiCORE IP Floating-Point Operator*. `http://www.xilinx.com/products/intellectual-property/FLOATING_PT.htm`.

[94] L. Yang, S. Gurumani, D. Chen, and K. Rupnow. Behavioral-level ip integration in high-level synthesis. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 172–175, Dec 2015.

[95] K. Campbell, L. He, L. Yang, S. Gurumani, K. Rupnow, and D. Chen. Debugging and verifying soc designs through effective cross-layer hardware-software co-simulation. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 7:1–7:6, New York, NY, USA, 2016. ACM.

[96] VERILATOR. *Verilator - Convert Verilog code to C++/SystemC*. `http://www.veripool.org/projects/verilator/wiki/manual-verilator`.

[97] 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.

[98] J. Curreri, G. Stitt, and A. George. High-level synthesis techniques for in-circuit assertion-based verification. In *IPDPSW*, pages 1–8, 2010.

[99] Y. Kim. *Formal Verification of High-level Synthesis with Global Code Motions*. PhD thesis, Syracuse, NY, USA, 2007. AAI3281725.

[100] T. Hong, Y. Li, S. B. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra. Qed: Quick error detection tests for effective post-silicon validation. In *2010 IEEE International Test Conference*, pages 1–10, Nov 2010.

[101] *LLVM's Analysis and Transform Passes*. `http://llvm.org/docs/Passes.html`.

[102] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin. GAUT: A High-Level Synthesis Tool for DSP Applications. In P. Coussy and A. Morawiec,

editors, *High-Level Synthesis*, chapter 9, pages 147–169. Springer Netherlands, Dordrecht, 2008.

[103] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau. Using global code motions to improve the quality of results for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):302–312, Feb 2004.

[104] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: a high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461–466, Jan 2003.

[105] S. Kurra, N. K. Singh, and P. R. Panda. The impact of loop unrolling on controller delay in high level synthesis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 391–396, San Jose, CA, USA, 2007. EDA Consortium.

[106] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4):441–470, October 2004.

[107] J. Cong, B. Liu, R. Prabhakar, and P. Zhang. *A Study on the Impact of Compiler Optimizations on High-Level Synthesis*, pages 143–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[108] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson. The effect of compiler optimizations on high-level synthesis for fpgas. In *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '13, pages 89–96, Washington, DC, USA, 2013. IEEE Computer Society.

[109] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In E. Sentovich, editor, *DAC*, pages 433–438. ACM, 2006.

[110] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of asics. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 8(6):661–679, 1989.

[111] R. Sinha and H. D. Patel. synasm: A high-level synthesis framework with support for parallel and timed constructs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(10):1508–1521, Oct 2012.

[112] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, December 1974.

[113] P. G. Paulin and J. P. Knight. Scheduling and binding algorithms for high-level synthesis. In *Design Automation, 1989. 26th Conference on*, pages 1–6, June 1989.

[114] R. Jain, A. Mujumdar, A. Sharma, and H. Wang. Empirical evaluation of some high-level synthesis scheduling heuristics. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, DAC '91, pages 686–689, New York, NY, USA, 1991. ACM.

[115] R. Camposano. Path-based scheduling for synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(1):85–93, Jan 1991.

[116] S. Bhattacharya, S. Dey, and F. Brglez. Performance analysis and optimization of schedules for conditional and loop-intensive specifications. In *Proceedings of the 31st Annual Design Automation Conference*, DAC '94, pages 491–496, New York, NY, USA, 1994. ACM.

[117] G. Lakshminarayana, K. S. Khouri, and N. K. Jha. Wavesched: a novel scheduling technique for control-flow intensive designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(5):505–523, May 1999.

[118] J. Cong, B. Liu, and Z. Zhang. Scheduling with soft constraints. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 47–54, New York, NY, USA, 2009. ACM.

[119] H. Zheng, S. Gurumani, L. Yang, D. Chen, and K. Rupnow. High-level synthesis with behavioral level multi-cycle path analysis. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, Sept 2013.

[120] M. Rim, R. Jain, and R. D. Leone. Optimal allocation and binding in high-level synthesis. In *Design Automation Conference, 1992. Proceedings., 29th ACM/IEEE*, pages 120–123, Jun 1992.

[121] R. Beidas and J. Zhu. Scalable interprocedural register allocation for high level synthesis. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, volume 1, pages 511–516 Vol. 1, Jan 2005.

[122] P. Brisk, A. K. Verma, and P. Ienne. Optimal polynomial-time interprocedural register allocation for high-level synthesis and asip design. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 172–179, Nov 2007.

[123] A. Mujumdar, M. Rim, R. Jain, and R. D. Leone. Binet: an algorithm for solving the binding problem. In *VLSI Design, 1994., Proceedings of the Seventh International Conference on*, pages 163–168, Jan 1994.

[124] H. Tomiyama, A. Inoue, and H. Yasuura. Statistical performance-driven module binding in high-level synthesis. In *System Synthesis, 1998. Proceedings. 11th International Symposium on*, pages 66–71, Dec 1998.

[125] D. Chen and J. Cong. Register binding and port assignment for multiplexer optimization. In *Design Automation Conference, 2004. Proceedings of the ASP-DAC 2004. Asia and South Pacific*, pages 68–73, Jan 2004.

[126] J. Cong and W. Jiang. Pattern-based behavior synthesis for fpga resource reduction. In *Proceedings of the 16th International ACM/SIGDA Symposium*

on Field Programmable Gate Arrays, FPGA '08, pages 107–116, New York, NY, USA, 2008. ACM.

[127] J. Cong, H. H. 0001, and W. Jiang. A generalized control-flow-aware pattern recognition algorithm for behavioral synthesis. In *DATE*, pages 1255–1260. IEEE, 2010.

[128] J. Cong, W. Jiang, B. Liu, and Y. Zou. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Trans. Des. Autom. Electron. Syst.*, 16(2):15:1–15:25, April 2011.

[129] L. Benini, A. Macii, and M. Poncino. A recursive algorithm for low-power memory partitioning. In *Low Power Electronics and Design, 2000. ISLPED '00. Proceedings of the 2000 International Symposium on*, pages 78–83, 2000.

[130] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong. Improving high level synthesis optimization opportunity through polyhedral transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 9–18, New York, NY, USA, 2013. ACM.

[131] T. Risset and A. Plesco. Coupling Loop Transformations and High-Level Synthesis. In *Symposium en Architecture de machines (Sympa 2008)*, Fribourg, Switzerland, 2008.

[132] W. Wang, T. K. Tan, J. Luo, Y. Fei, L. Shang, K. S. Vallerio, L. Zhong, A. Raghunathan, and N. K. Jha. A comprehensive high-level synthesis system for control-flow intensive behaviors. In *Proceedings of the 13th ACM Great Lakes Symposium on VLSI*, GLSVLSI '03, pages 11–14, New York, NY, USA, 2003. ACM.

[133] A. Raghunathan and N. K. Jha. Scalp: an iterative-improvement-based low-power data path synthesis system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(11):1260–1277, Nov 1997.

[134] H. Zheng, S. T. Gurumani, K. Rupnow, and D. 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.

[135] W. Wang, A. Raghunathan, N. K. Jha, and S. Dey. High-level synthesis of multi-process behavioral descriptions. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 467–473, Jan 2003.

[136] R. Mukherjee, S. O. Memik, and G. Memik. Temperature-aware resource allocation and binding in high-level synthesis. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 196–201, June 2005.

[137] H. Zheng, S. Gurumani, L. Yang, D. Chen, and K. Rupnow. High-level synthesis with behavioral level multi-cycle path analysis. In *FPL*, 2013.

[138] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *FPGA*, pages 33–36, 2011.

[139] *Clang: A C language family frontend for LLVM.* `http://clang.llvm.org/`.

[140] J. Zhang, Z. Zhang, S. Zhou, M. Tan, X. Liu, X. Cheng, and J. Cong. Bit-level optimization for high-level synthesis and fpga-based acceleration. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pages 59–68, New York, NY, USA, 2010. ACM.

[141] *LLVM Alias Analysis Infrastructure.* `http://llvm.org/docs/AliasAnalysis.html`.

[142] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *DAC*, pages 433–438, 2006.

[143] H. Zheng, Q. Liu, J. Li, D. Chen, and Z. Wang. A gradual scheduling framework for problem size reduction and cross basic block parallelism exploitation in high-level synthesis. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 780–786, Jan 2013.

[144] H. Zheng, S. Gurumani, L. Yang, D. Chen, and K. Rupnow. High-level synthesis with behavioral-level multicycle path analysis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 33(12):1832–1845, Dec 2014.

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

[146] A. Freitas. Hardware/software co-verification using the systemverilog dpi. `http://www.qucosa.de/fileadmin/data/qucosa/documents/5410/data/13_Freitas.pdf`.

[147] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009.

[148] W. J. Townsend, J. A. Abraham, and E. E. Swartzlander, Jr. Quadruple time redundancy adders. In *DFT*, pages 250–. IEEE Computer Society, 2003.

[149] Synopsys, Inc. *Siloti Visibility Automation System*. `https://www.synopsys.com/Tools/Verification/debug/Pages/siloti-ds.aspx`.

[150] L. Yang, S. Gurumani, D. Chen, and K. Rupnow. Autoslide: Automatic source-level instrumentation and debugging. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24rd Annual International Symposium on*, May 2016.

[151] *LLVM Language Reference Manual*. `http://llvm.org/docs/LangRef.html`.

[152] *Source Level Debugging with LLVM.* `http://llvm.org/docs/SourceLevelDebugging.html`.

[153] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *ISCAS*, pages 1192–1195, 2008.

[154] D. Lin, T. Hong, F. Fallah, N. Hakim, and S. Mitra. Quick detection of difficult bugs for effective post-silicon validation. In *DAC*, pages 561–566, 2012.

[155] *Kaleidoscope: Adding JIT and Optimizer Support.* `http://llvm.org/docs/tutorial/LangImpl4.html`.