grind: A Framework for Dynamically Instrumenting and Verifying HLS-generated RTL

by

Parmida Vahdatniya

B.Sc, Sharif University of Technology, 2018

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

Declaration of Committee

Name:	**Parmida Vahdatniya**
Degree:	**Master of Science**
Thesis title:	**grind: A Framework for Dynamically Instrumenting and Verifying HLS-generated RTL**
Committee:	**Chair:** Mo Chen Assistant Professor, Computing Science
	Arrvindh Shriraman Supervisor Associate Professor, Computing Science
	Alaa Alameldeen Committee Member Associate Professor, Computing Science
	Zhenman Fang Examiner Assistant Professor, Engineering Science

Abstract

High-level synthesis (HLS) compilers enable the rapid creation of custom accelerator circuits [54, 32, 37, 42]. However, HLS-generated RTL (H-RTL) is inconsistent in terms of quality, too verbose to be comprehensible, and may even have functional errors [38, 26]. We propose a framework that helps designers inspect, instrument, and profile H-RTL. State-of-the-art tools such as [22, 56] have predominantly focused on tracing. Unfortunately, tracing requires a massive amount of memory, limits the H-RTL size, allows for faults to propagate to other modules, and expects the user to manually identify the signals. Further, the tools can only run post-execution [24, 21, 23] which limits the types of analysis the designer can perform.

In this thesis, we propose grind, a dynamic instrumentation framework that enables computer architects to observe, and modify signals during the execution of the accelerator prototype. The key technique is guards, additional circuits that we automatically attach to the H-RTL (without requiring human intervention for insertion or removal). Guards perform two activities: i) Run analysis functions on the values fed from the H-RTL signal. ii) Inject values into registers, wires, and memory entries of the H-RTL and patch the execution. During prototyping guards get mapped onto the FPGA along with the H-RTL; grind removes the guards once the H-RTL is finalized. We use guards to develop a verifier tool that instruments the H-RTL iteratively and locates a faulty module. Compared to state-of-the-art [56], We also introduce two additional tools: i) *H-RTL Faulty*, which uses guards to inject faulty values and observe the propagation of erroneous values in the circuit, and ii) *H-RTL profiler*, a lightweight guard for profiling the data values, hardware signals, and addresses. We require between 200-35000X less DRAM traffic than off-chip profilers

Keywords: Accelerator; RTL Verification; Hardware debugging

Dedication

To Helia

Acknowledgements

First I would like to sincerely thank my supervisor, Dr. Arrvindh Shriraman, for his wonderful support and guidance during my studies and research, and for the opportunity to work under his supervision.
Many thanks to my examining committee, Dr. Alaa Alameldeen, Dr. Zhenman Fang, and Dr. Mo Chen.
I'm very thankful to Amirali and Reza for their extensive support, and to Ali and Minh. It was a great pleasure to work with all of you. And I would like to thank the fellow students of the SFU Computer Architecture lab, whom I had the pleasure of working with.
Many thanks to my amazing family and friends, Rajab, Mona, Sepid, Keivan, and Sara. And finally, I am thankful to my incredible mother to whom I owe a great debt of gratitude.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years there has been a surge of research in high-level-synthesis compilers (HLS) that auto-translate high-level languages [31, 30, 53, 54, 52, 41, 45, 16, 47, 5]. High-Level Synthesis (HLS) promises improved designer productivity by allowing designers to create digital circuits on FPGAs. However, the widespread adoption of HLS tools is limited by the lack of on-chip verification ecosystems that bridge between the software and the generated hardware. Computer architects broadly acknowledge that the primary challenge in HLS is the opaqueness and inaccessibility of the generated RTL (H-RTL) [38]. Further, even industry-standard HLS compilers are known to break the H-RTL in eccentric ways due to incorrectly specified pragmas [26]. The challenges have been further compounded by the plethora of HLS compilers targeting both fixed-function and reconfigurable hardware [31, 30, 53, 54, 52, 41, 45, 16, 47, 34, 7, 5]. Recent HLS compilers translate semantically rich code into hardware, including tasks [41], nested control[32], irregular memory accesses, and nested parallelism [37]. This, coupled with the fact that HLS compilers also include an extensive set of optimization passes makes the final H-RTL opaque to the designer and challenging to manually verify. [13, 54, 32, 48].

1.1 Challenges of working with HLS

A leading HLS expert cites the lack of mature tools to inspect H-RTL as being a key hindrance [13, Page 8] [38] to HLS adoption. Multiple tools exist (e.g., Valgrind, Dynamorio, gcc -p) to analyze the output of software compilers (i.e., binaries) on CPUs. However, H-RTL lacks such a framework.

The most widely used approach to verifying HLS-generated RTL (H-RTL) is comparing the final memory state against the expected values (gathered from the software) after the simulation and run, the comparison is either done manually by the user or automatically and the first place of mismatch is reported to the user. However, this method is woefully inadequate: i) When a mismatch is identified, the designer must look through a long simulation trace and walk backward over time to narrow the cause, and ii) the process may require a detailed analysis of tens of thousands of signals, throughout thousands of simulation cycles. further, the primary suggestion from commercial vendors is to use waveforms [3] for verifying functionality. State-or-the-art works in academia have sought to supplement waveforms with execution traces obtained either through simulation or from an FPGA [24]

Waveforms are the most prevalent approach to analyzing HLS-generated RTL (H-RTL) [3]. This requires the user to inspect the H-RTL netlist through potentially millions of simulation cycles [35]. Due to the associated overheads, waveforms are typically collected using RTL simulation, not tractable for large designs. State-of-the-art tools help annotate waveforms with additional information gathered either through simulation or from an FPGA [25, 9, 20, 28, 56]. They enable the user to focus on a specific segment of the trace and waveform and place an excessive burden on the end-user to examine the trace. They leave open the question of **"how can a user know which portion of the H-RTL to focus on, how to catch multiple faults and failures, and how to deal with faults that cause the execution and run to fail?"** Some works have provided a gdb-like interactive environment [8, 12, 23, 29, 11]. The user has to markup the required software variables and statements, and the HLS compilers track the corresponding portions in the H-RTL. They adopt stop-the-circuit semantics (similar to an assert [51]) and are useful only for post-mortem. They do not permit progress past the assertion. This cannot support tools such as fault analysis where the tool needs the circuit to continue execution with faulty values propagating through the circuit. grind is a novel tool to tackle the problem of instrumenting H-RTL, while prior work has focused exclusively on post-execution analysis.

1.2 Why do we suggest a new H-RTL instrumentation tool?

The challenges that designers and users face while verifying H-RTL were mentioned in the previous section. The main thing we aim to achieve in this work is to help the designer instrument the HLS-generated RTL with minimum effort and flexibility, to understand the dynamic execution of hardware. Also, many designers may not be fully familiar with every aspect of the H-RTL, hence, an automated tool that only requires the user to be familiar with the data flow to run it can be extremely beneficial. In contrast to previous methods where the designer had to look through the traces of hardware execution (see Figure 1.2). The goals of this work are as follows:

- Instrumentation of the H-RTL with minimum to zero effort i.e., the designer should be able to read, analyze and write H-RTL signals without needing to edit the H-RTL manually or knowing a hardware language.
- Flexible and conditional instrumentation i.e., a configurable framework that adds in or removes additional logic and SRAM only for the signals instrumented in the H-RTL, allowing the framework to remove the instrumentation entirely from the H-RTL once the accelerator is analyzed as well.
- Dynamic instrumentation i.e., the instrumentation can analyze and modify signals **during** the execution. We demonstrate that live execution analysis is essential to creating verifier tools that avoid muddled-up logs.

1.3 The target platform

We target IR [54] as our platform in this work. IR is an intermediate representation for accelerator microarchitectures. IR operates as follows: i) A decoupled graph from the actual hardware components is generated, which decouples the microarchitecture optimizations from algorithm/program optimizations and the actual RTL ii) this graph is translated to Chisel, as intermediate hardware. iii) Finally the Chisel representation translates into FIRTLL and Verilog. The execution model and cycle-level performance are maintained.

IR graph represents the microarchitecture components of the hardware and the data transition between these components. Since this graph is written in Chisel, which is at a higher level than RTL, optimizing the accelerator microarchitecture becomes less complicated and easier [54].

1.4 Our approach

We introduce grind, a framework for dynamically instrumenting HLS-generated RTL. Figure 1.1 shows an abstract view of the concept of grind. We instrument both the intermediate software and hardware (Chisel.) The instrumentation of LLVM is done to extract helper data (Metadata) and the instrumentation of Chisel is done to either modify, inject or observe values in the hardware. The key technique of grind is guards, hardware modules that we attach and inject to the original H-RTL to tail and shepherd specific register, memory entry, and signals (Chisel instrumentations). grind builds on modern RTL compilers (Chisel and FIRRTL [27]) to add and remove guards. Guards are mapped onto the FPGA prototype with H-RTL; they can also be co-simulated in the verilator. During the execution, guards dynamically extract, run analysis logic, and update the H-RTL's signals. This eliminates the need to trace a verbose dump of signals to the DRAM for post-execution analysis, as guards can decide which data would be useful to record and which data to discard. Guards only need to write post-analysis data to the DRAM. This saves DRAM traffic, reduces the on-chip SRAM, and enables larger circuits to be analyzed. Further, the analysis functions run concurrently in hardware which reduces the overhead typically associated with instrumentation.

Figure 1.2 shows the toolflow of our approach to verify a circuit, the general idea is instrumenting the circuit, and in scenarios such as verification, doing it multiple times based on the reports of the previous instrumentation. This instrumentation allows the user to verify (amongst other side tools) the design in the way that they prefer, freely and more flexible. Further, we know that not every user working with an HLS framework is familiar with every piece of code of the framework as usually multiple designers work on a project, and editing the code manually to observe and manipulate the design could be a time-consuming task. We also realize that previous debugging tools face two main obstacles, memory and time. As mentioned above, we address the issue of memory overhead by only recording the signals that are useful for the user and storing as minimal values on memory as possible. We also propose a fully automated framework, meaning user interference is

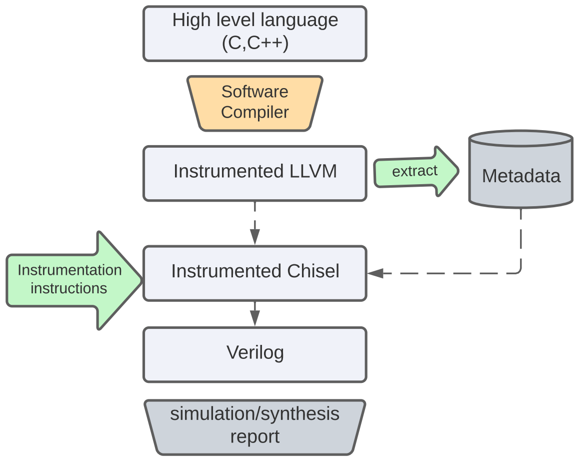


Figure 1.1: An abstract overview of the instrumentation of the intermediate software (LLVM) and the intermediate hardware (Chisel) to implement grind.

not needed after their first interaction with the framework. So time won't be as much of an obstacle as it's just a program running without needing human attention and time.

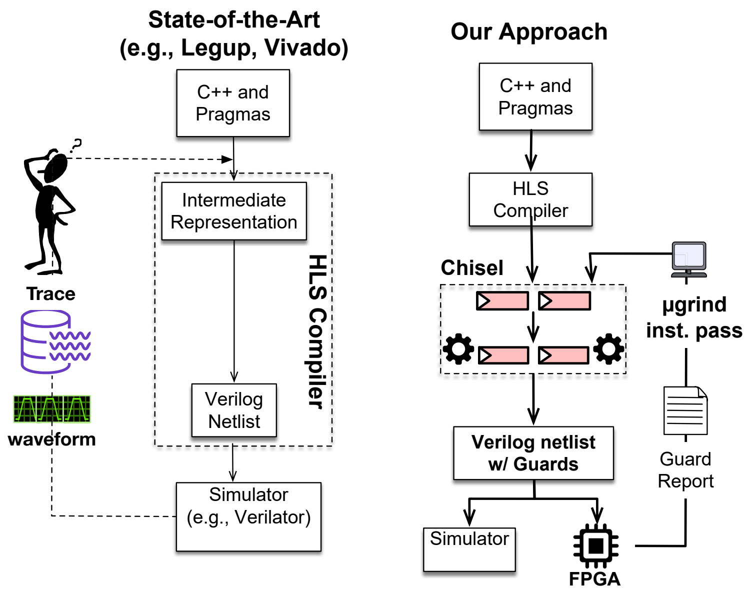


Figure 1.2: This figure provides a comparison between previous trace-based HLS verification toolflows and the toolflow of our approach.

We create an automated verifier and two other tools to demonstrate the use-cases of grind and guards. As shown in Figure 1.3 a) *H-RTL verifier:* a novel verifier, that pinpoints the RTL statements (alongside their cycles and any other additional information needed) in which the H-RTL deviated from the expected software values. grind verifier exploits automatic guard injection scripts (imple-

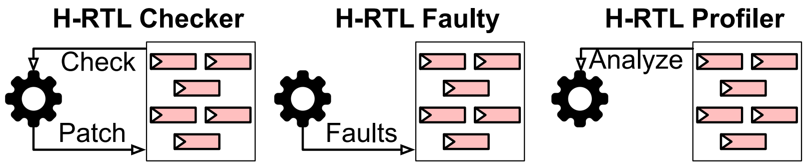


Figure 1.3:   grind Tools

mented using Chisel and FIRRTL [27] tools) to create an iterative and iterative verifier that searches through subsets of modules instead of the entire circuit and discards part of the circuit that behave correctly. b) *H-RTL Faulty :* leverages guards to inject a variety of faults (e.g, stuck-at-zero) into specific H-RTL signals, while simultaneously using guards to check other signals in the circuit. Faulty helps the user check the resiliency of the circuit. c) *H-RTL Profiler :* a tool that builds into hardware the logic for extracting and summarizing data during execution to avoid large data dumps to the DRAM. However, as we further demonstrate guards and   grind we learn that the use-cases of this instrumentation platform are not limited to these three tools.

The contributions of   grind are as follows:

- An open-source framework for dynamically instrumenting HLS-generated RTL (H-RTL) with guards (in many ways). FPGA synthesis reveals that guards impose limited overheads, 10—15% extra logic, and   5% Mhz penalty.
- An iterative verifier (one of the instrumentation techniques), that can identify and report hardware errors and help HLS compiler research. We study complete accelerators [50, 49] and demonstrate that: i) we check circuits 5   larger than prior state-of-the-art, and ii) we can complete verifying in under 24hrs (including FPGA synthesis) and less than 4 iterations (less than 2hrs in simulation).
- Two additional tools, H-RTL faulty, for studying fault resiliency and, H-RTL profiler, for collecting live statistics. These tools demonstrate the benefits of guards that can analyze live execution without needing to dump signals to the DRAM for post-mortem. We save 200—35000   DRAM traffic and 2—10   of on-chip SRAM.

We next talk about H-RTL instrumentation and the challenges of it, errors and faults that can happen in H-RTL and the ones we look at and verify in the evaluation, the benchmarks used to verify and evaluate our work and their complexity, and prior works done in the field in **Background and Related work**.

Next, we look into the design and implementation of Guards in **Architecture and Design**. The first step in the design is to instrument the software to extract the DFG values and IDs and the dependence graph information for the verifier tool. Next, we look into the "Boring" tool and how it allows us to automatically and conditionally wire guard modules into the main circuit. And finally the internal design of guards and the wrappers that attach them to the original circuit.

In the **Verifier Tool** chapter we look at our main Grind tool. We discuss the idea of iteratively verifying the design and how the automation of guards helps us do this, the algorithm behind the verifier, and a working example.

We then look into two additional tools built using Grind in **Additional Tools**. H-RTL profiler is a smart profiler that dynamically profiles specific data and H-RTL faulty is a fault injector which allows the designers to test their design under potential faults.

Finally, we evaluate the introduced tools in **Evaluation** we first inject bugs into the benchmark circuits and catch them using the verifier. We further look into an actual bug that we traced using the iterative verifier as a real-life use case. Next, we evaluate the profiler and fault injector tools by reporting memory usage and observed outcomes of the injected faults.

Further implementation discussion and instructions of using the platform are presented in the **Appendix**.

Chapter 2

Background and related work

In this chapter, we present a quick overview of the [54] dynamic dataflow platform and the challenges of H-RTL instrumentation and the motivation behind Grind. We next introduce several possible H-RTL faults and errors that we examine and the benchmarks we use to test our tools. We finally learn that the motivation behind grind is the manual steps that the user had to take to find and fix faults in [54].

2.1 Dynamic Dataflow

HLS tools such as [54] construct circuits with dataflow components. Such components typically rely on an asynchronous mechanism to exchange data, or "tokens". We choose IR [54] as our target platform. The token exchange protocol in IR uses two signals i) one signaling the "availability" of a new token from the source component ii) the other signaling that the target component is "ready" to accept it. This protocol is called "Handshaking". The Handshaking implementation in the components is shown in Figure 2.1. In contrast to a predetermined, centralized controller of statically scheduled circuits. This asynchronous control allows dataflow circuits to adapt the schedule at runtime and take into consideration the latency of certain memory access patterns and control-flow decisions. In addition to standard functional units, dataflow circuits require specialized components which control the flow of data between components.
The advantage of using IR as an HLS framework compared to the other HLS frameworks for us is that the hardware components are implemented in Chisel, and we can plug our guard modules as a hardware pass to the main design using "Boring" Connections from FIRRTL[27], this is further discussed in the following chapters. However, grind is not limited to only one HLS framework, and it can work with other HLS frameworks that provide a structure similar to [54]. Figure 2.1 outlines the dataflow components that are common in such designs and in IR.

A summary of these components:

- *Compute Node:* applies the compute function and replicates every token received at the input to multiple outputs; it outputs tokens to each successor as soon as possible but does not accept any new token until all successors have accepted the previous one.

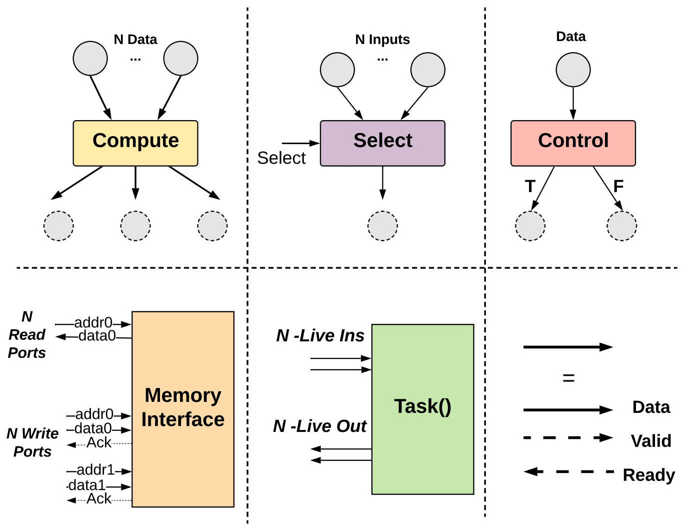


Figure 2.1: Dynamic Dataflow Components

- *Mux Node:* waits for the required input to produce the output and discards the tokens at the non-selected inputs as soon as they arrive.
- *Control Node:* implements program control-flow statements (i.e., if or switch) by dispatching a token (and, sometimes, the corresponding piece of data) received at its single input to one of its multiple outputs based on a condition.
- *Memory:* The system interfaces with memory through handshake ports. The write port has two inputs (data and address) and a control-only signal from the memory interface indicates successful completion. The read port sends an address to memory and receives data with its corresponding handshake control.
- *Task:* Each task contains a set of *live-in*s, *live-out*s and local tasks queue that stores ready and pending tasks. The task is free to process the ready tasks in any order. Each parent task spawns children to run concurrently and children terminate and return values to the parents. Tasks communicate either through memory or through registers in the connection.

Next, we look into how software programs are translated to dataflow circuits [54, 32, 13]. In the absence of control flow and concurrency i.e., a single basic block the translation process is simple. The HLS compiler simply takes the data dependencies between the operations and creates a dataflow circuit. The abstraction is broken only at the memory interface. Different HLS compilers treat memory differently. Legup [13] serializes access to global memory, Dynamatic [32] introduces a centralized load-store queue, and -IR [54] introduced a distributed queue. HLS tools implement control flow through either trigger instructions or predication. Typically simple diamond-like control patterns are converted into predicated dataflow graphs. Some HLS compilers support more sophisticated control flow. In such cases, the HLS tool processes the compiler program dependence graph to determine the basic blocks and assign a trigger register to each basic block. The trigger register essentially fans out to the operations in the basic block and serves as a start signal. We then implement the control flow graph by connecting the labels to the respective branch

In this accelerator microarchitecture there is two primary class of bugs that could be introduced i) the HLS compiler itself may optimize away a dependency or mismanage the tokens that need to be exchanged between data-dependent operations resulting in lost data. ii) the HLS compiler typically includes a library of black-boxed microarchitecture components (e.g., for FPUs, caches). A bug could have been introduced in the RTL of these components. These components are invisible to the designer since they are introduced by the compiler during the auto-synthesis process. These are particularly challenging as the bug may originate at an invisible point in the RTL, but may manifest in an operation thousands of cycles later. We will discuss the types of bugs we look into in this work further in this chapter.

Back to the original question: *What should a designer do to capture a potential set of faults in the implementation of each of these components, or the connections between them?* The steps are quite time-consuming and challenging especially for a designer that is not necessarily familiar with every registered name and component implementation. They have to enable a trace-based profiler, either in the form of a waveform or printing/storing trace values. The designer then either has to find the first place the trace behaved wrong manually or, if they are using a more advanced tool such as [24], they will be informed of it. However, this method i) won't work if the fault had caused the circuit to fail ii) it will take a very large amount of memory iii) if there are multiple faults in the path they won't be caught unless the designer fixes the fault that came before them. Moreover, grind can do more than report the faults, as it is originally an instrumentation tool, and if the user for example wants extra information about each fault, they can manipulate the guard code to gather it.

2.2 Why do we need an instrumentation tool?

Any large software project is prone to bugs and code rot, HLS compilers are no exception. Industry-standard HLS tools (particularly the internals) have largely been under the closed source. Only recently, have there been efforts to systematically document bugs. Herklotz et al. reported between 0.5% to 3% of C microbenchmark suite failed across multiple versions of the industry-standard HLS tools [26]. Commercial HLS tools have primarily focused on achieving the best performance, area, and energy tradeoffs, and have often overlooked correctness. *HLS bugs are difficult to identify and exist because it is not clear to the user how the generated design behaves.* HLS users implicitly assume that the RTL is functionally equivalent to the program, but there is no certain way to validate this. The most common approach has been testing and tracing the output memory. However, this is heavyweight and test benches inherently miss out on circuit regions. There have been formal approaches to prove compiler correctness. [36, 40]. They typically turn off specific compiler optimizations and this results in designs with higher area and power penalties.

Further, we highlight the unique challenges of instrumenting H-RTL. We illustrate how dynamic instrumentation can help catch H-RTL errors introduced by a state-of-the-art HLS compiler [54]. Next, we quantify the complexity of H-RTL circuits we study and motivate the need for automated tools and finally introduce the types of faults we look into.

2.3 H-RTL instrumentation vs. Binary instrumentation

HLS developers cite the lack of fixed semantics and state as to why H-RTL is harder to instrument [6, 39]. **i) Executable (binary) vs. Structural (H-RTL):** A binary runs on existing hardware. Instrumenting the binary entails adding instructions to read and write the registers/memory. These instrumentation instructions run on the same hardware interleaved with the binary. Adding instrumentation edits the structure of H-RTL and is more involved. We have to allocate additional logic, bind operations to the logic, and physically route values. **Implication :** Need automation and scripting to edit the H-RTL and mix in the instrumentation, after the HLS.

ii) Imperative ISA (binary) vs Concurrent Dataflow (H-RTL) A binary is an imperative specification in an ISA defined by the underlying processor target. Further, the instrumented binary implicitly supports sequential semantics enforced by the underlying CPU. H-RTL is a concurrent specification in which the timing and order of operations have to be defined by grind. **Implication:** Need a flexible approach to identify H-RTL events that trigger the guards and enforce the ordering between the guards and H-RTL signals.

Centralized fixed state (binary) vs Distributed, variable state (H-RTL) Finally, any instrumentation framework needs to read and write state from the target `https://blog.regehr.org/archives/1450`. With binary, the architectural registers and memory state is defined, centralized, and accessed via the processor instructions. All binaries refer to the same architectural registers. HLS customizes the state for each H-RTL circuit and distributes the state across the pipeline latches, operand buffers, and scratchpads. **Implication:** grind needs to maintain the software to hardware mapping so that the instrumentation reports can be presented to the user at a higher level. We also need to support the removal of instrumentation when not required.

2.4 H-RTL errors and faults

In this section, we look into the possible bugs that can show up in H-RTL by tracking the git commits in a state-of-the-art HLS compiler [54](IR). We demonstrate how instrumentation can track signals, variables, and cycle timing to catch underlying errors in H-RTL. We communicated with the authors of IR and verified the cause of these errors [14, 41, 54].

H-RTL Error 1: Stuck-at-zero merge node
github@muir/SelectNode.scala hash:#3d51bcb301.
Detection: Instrument the merge mux's output signal.

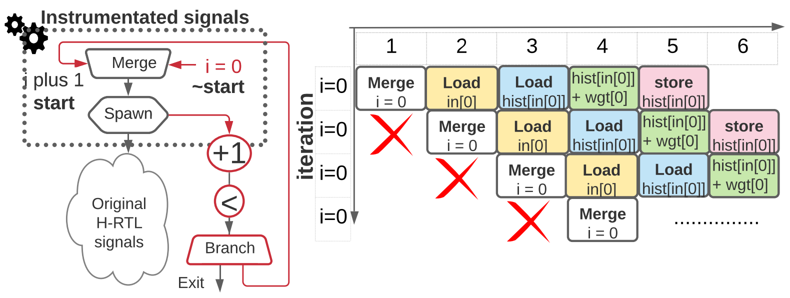


Figure 2.2: H-RTL Error 1: Stuck-at-zero merge node caused by LLVM syntax mismatch leading to incorrect mux wiring.

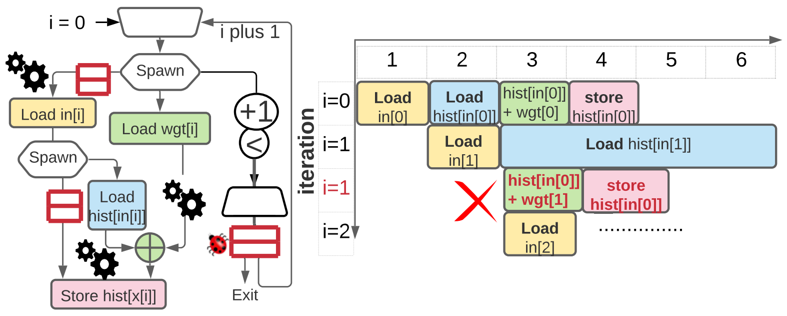


Figure 2.3: H-RTL Error 2: Incorrect pipeline buffer depth setting leading to faulty operands.

Many HLS compilers translate LLVM's SSA representation to RTL (e.g., LLVM IR [54, 32, 2]). LLVM periodically updates the SSA syntax during major releases. In this instance, LLVM reversed the order of labels in the select and ops. This led the HLS to wire the mux data lines to the merge node in the incorrect order. Due to the mix-up, the mux is stuck at and always propagates i=0 on each iteration of the loop; the loop keeps re-executing iteration i = 0. Tracing of waveforms cannot catch this bug since execution will never terminate. grind's dynamic instrumentation will capture the output of the merge signal and the analysis will check if the values of the signal are incrementing like a loop induction variable.

H-RTL Error 2: Incorrect dataflow pipelining

github@muir/LoopBlock.scala hash:#a4245dd02f

Detection: Instrument the output signal of dataflow operators and check against SSA register values.

These classes of errors are reported even by Xilinx's Vivado [55]. HLS compilers place FIFO buffers to i) enable loop iterations to start asynchronously, and ii) to balance the different critical paths at spawns. In this instance, the HLS compiler miscalculated the latency of paths and created a buffer with incorrect depth. As shown in the timing diagram this leads to incorrect operands being placed on the inputs to the adder; one of the operands is from the ith iteration and the other one from i-1th. Dynamic instrumentation will track the values in the output registers of the nodes, check the

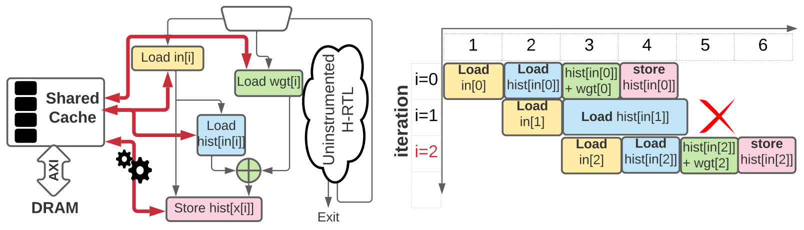


Figure 2.4: H-RTL Error 3: Incorrect interfacing between H-RTL and Cache leading to missed request and circuit lockup.

iteration index and the operands of the adder.

HLS Error 3: Faulty Cache Handshaking github@muir/Cache.scala #ea42742eaed
Detection: Instrument the cache request and response lines, and check number of requests/responses.

A common cause of the error is the interface to the shared cache (or scratchpad). Typically the cache or scratchpad is a black box IP invisible to HLS. The HLS statically schedules loads and stores across latency-sensitive request and response ports. In this particular case, IR HLS incorrectly scheduled the load(in[i]) on the same cycle as another load. This led to a load being missed by the cache. IR HLS [54] also reported similar errors causing incorrect response errors due to wrong address. grind instruments the cache request and response lines along with the memory nodes. It analyzes the sequence of requests and responses to verify if every request has a corresponding response. These types of verifiers can be since grind permits the user to define analysis function within the guards.

Bug Example 4: Blackbox IPs.

Reported: github@muir/CustomComputeNode.scala –commit(cbe1845260e)

Not all RTL modules in the prototype are auto-generated by the HLS. Typically the prototype and final tape out include multiple BlackBox IPs (e.g., technology PDK-specific RAM modules, Floating point hardware). The interface to the BlackBox modules is exported to the HLS compiler along with the timing constraints of the individual ports. In this particular bug, a single configuration parameter had led to IR-HLS making incorrect assumptions about the pipeline depth. This led to operands being initiated for the BlackBox earlier than specified and a breakdown of the pipeline. The challenge with these bugs bug may originate at a point in the RTL invisible to the HLS compiler but may manifest in an operation thousand of cycles past the statement in H-RTL.

2.5 Complexity of instrumented H-RTL circuits.

We study end-to-end applications from Machsuite [50] *Relu, Saxpy, Vadd, Conv2D, Stencil, and Gemm*. Table 2.1 lists the characteristics of the H-RTL circuits. The H-RTL can be viewed here

App.	Verilog LOC	# FSM	# Verilog Mod.	Pipe. Depth	Parallel
GEMM	33049	32	366	14	32
Conv2D	37277	16	329	41	48
FFT	37418	4	340	22	56
Relu	21051	4	206	11	48
Saxpy	18060	2	228	9	48
Stencil	26396	8	166	8	768

Table 2.1: RTL Complexity of guarded Accelerators studied in this thesis.

(https://anonymous.4open.science/r/d6f70aaf-3014-4353-9b48-cc5759080898/). In this paper, we study accelerators and varied types of nested parallelism including loop, data, and instruction parallelism. Prior works only studied H-RTL circuits with unrolled loops [29, 56].

Since there is no standard metric to quantify RTL-complexity, we use four proxy metrics to provide intuition on the complexity of the designs we study i) **Verilog LOC:** The number of lines of RTL code in each of the circuits; reasonable proxy for the number of H-RTL variables (signals or registers). ii) **Ctrl-states** this is indicative of the complexity of the FSM of the circuit. Typically this is lower for token-driven dataflow circuits that do not use a global FSM. However, in kernels with multiple nested loops, FSMs are required to coordinate the interactions between the loops. iii) **Verilog modules:** The total number of modules instantiated in the code. Higher the number of modules the more effective grind is for narrowing the site of an error. iv) **Pipeline depth:** In HLS, the pipeline structure varies across accelerators. Here, we can see that in some of our workloads the pipelined depth can be 50 stages (well beyond a conventional processor). This is indicative of the challenge of instrumenting and analyzing timing-dependent errors that may show up only in a specific cycle and pipeline register. v) **Concurrency:** Finally, we measured the number of concurrent operations on the hardware datapath. The H-RTL circuits we investigate are highly concurrent; an instrumentation framework is required to track down timing errors.

2.6 Related Work

Prior tools do not support generalized instrumentation and user-defined tools. A key difference compared to our work is the **target and type of instrumentation.** And also flexibility to manipulate the checker tool to perform different sorts of verification. Checker tools tailor instrumentation towards the C/System-C input. These tools require human-in-the-loop to manually identify the scope. Further, many of these tools use trace-based approaches that limit the on-chip memory. And hence, they always face the decision between limiting memory, or optimizing memory usage and losing verification data.

[22, 20] points out the lack of debugging tools for HLS and the problem of not having visibility into circuits. Targeting both Hardware and Software designers, they use tracing buffers to trace the circuit run and capture circuit values to map back to software. However, tracing will take a large

amount of memory especially since during the run the user will not know which part of the circuit is faulty. They acknowledge this and look into ways to optimize the trace buffers and on-chip memory.

[8] Further discusses the issue of memory usage vs losing data. Pointing out that since memory is limited most of the other trace-based methods tend to ask the user to select a number of variables to trace. But the user does not know which variables may be buggy, or any other information to help them with the decision. This usually leads to multiple runs. So they propose a method to accelerate the debug turnaround time.

[43, 44] Present another trace-based debug approach by introducing observability ports and buffers, the ports are used to observe the variables in the circuit and buffers to trace and record them. The novelty of this approach is that the user is permitted to make tracing decisions (whether to store data in a buffer) based on the values they observe. They point out that this method causes the possible loss of timing relationships of events for different trace buffers. They also need to keep the user in the loop to make the tracing decision. They propose a method that eliminates timing, latency, or throughput being affected by their observability tool.

[56] Autoslide, which is the work we evaluate against the most is an automated cross-layer verification framework. Similar to our approach Autoslide indicates the importance of automation in debugging and focusing on certain critical operations first. Autoslide is however another trace-based method and suffers from most disadvantages that other trace-based debuggers do. More comparison to Autoslide is shown in Table 6.4. Autoslide also maps the RTL datapath to LLVM-IR operations and C/C++ source code (as we do as well) to minimize user effort.

[9] proposes a gdb-like debugging tool for HLS. They also allow the user to map the HLS values back to software representation. Supporting both simulation and execution on FPGA, the framework aims to provide RTL values for each C statement. Similar to other trace-based tools this approach allows the user to only view the first place of mismatch.

Another common tool HLS designers use is Assertions which are included in the H-RTL at specific signals and kill the run once activated. Assertions put the responsibility of figuring buggy locations on the user as it's the user who must decide where to insert them. Further, asserts typically check a fixed condition e.g., signal == 0?. They cannot accommodate value-based checks, tracing, and collecting data the same way debuggers and checkers do. Finally, assert triggers only at deviating signal; the error may have propagated from the non-assert location.

[33] is an open-source performance profiler that uses out-of-band call stack reconstruction and performance counters that we compare against. Some of them target hand-written RTL [33] not verbose HLS-generated RTL. **Value-based** Prior art does not permit the user to check if the check depends on the actual value of the H-RTL signals. The exception being simulation-based approaches [56] that permit use of printfs(). All the value profilers and checkers we have demonstrated in this paper need to extract the signal values; prior art cannot implement them. **Low-effort and Autogen.** The effort required to insert instrumentation into the H-RTL impacts utility. For instance, without low-effort instrumentation, human intervention is needed to decide where and what to instrument. The majority of prior art lacks a flexible mechanism [22, 20, 56]. **Execution analysis**

In prior work, the analysis of the signals is postponed to post-execution. They rely on waveform extraction, which incurs a significant bandwidth penalty

Note that closed-source formal RTL checkers are only loosely connected to this paper. Catapult HLS employs SLEC [7] is a form of logic checking for analyzing H-RTL. Logic checking and translation are computationally intensive. It has only been demonstrated on circuits as complex as FP ALUs and even that requires 12 hrs [46] per FPU. We demonstrate that we can identify bug sites in end-to-end accelerators (e.g.Convolution) in 20 hrs. Further, SLEC only works with H-RTL circuits that are finite-state-machine with datapath [4] and support sequential semantics only. SLEC cannot be applied to the HLS compilers we target. They generate circuits with concurrent dataflow semantics, dynamic parallel patterns, and non-deterministic global memory accesses [32, 54, 52, 47, 15].

Chapter 3

Architecture and Design

In this chapter, We look into the general toolflow of grind shown in Figure 3.1 and the implementation of grind. We first go through the process of extracting data from the intermediate software, this data will be used by various grind tools to analyze the H-RTL values. Next, we describe the guard design and how we generate guards for the intended signals and connect them to components automatically and conditionally. Finally, we describe the architectural template that integrates with the H-RTL circuit and gathers data during execution. But first let's answer a general question: *What is a guard?* A guard is a hardware module meant to attach to circuit components and "guard" them. The "guarding" process is based on the guard function which the user will define, it can be anything from just observing and recording signals and values to processing and changing them. The default guard function in this work is set to the verifier-guard, meaning that the guard will compare the outputs of modules against the expected outputs and if they didn't match a flag is set to true and the value is corrected. a Guard module consists of two buffers, one for incoming data and one for outgoing, plus a "guard function" that performs analysis on the input data and the data observed from the hardware component it's attached to and writes the results of the analysis on the output buffer.

3.1 IR Metadata

First, we look into the verifying tool to check correct execution, we can fundamentally characterize correct execution with a few properties: input data received, output data produced, conditional control transitions, the correctness of data propagation through select operations, and forward progress in execution.

We describe grind verifier designed to identify functional bugs, these types of bugs are the result of defective hardware units and connections, either caused by the compiler or the actual implementation. grind verifier aims to provide the user with a clear notion of the places in which the hardware behaved incorrectly, to achieve this, we first need to have a baseline as the expected correct behavior to compare against. So let's assume at first that we can observe and compare the outputs of all hardware components.

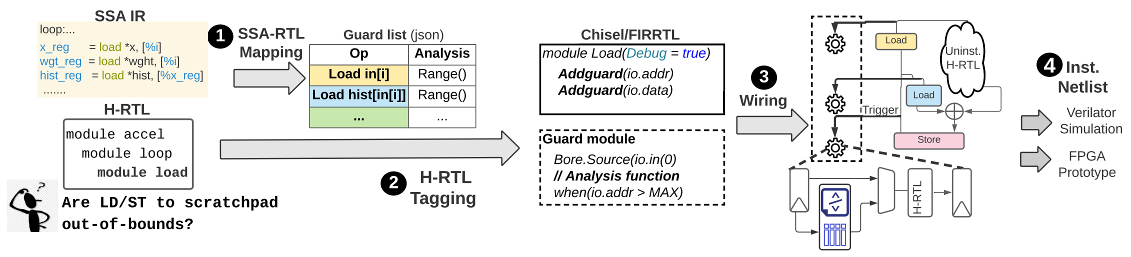


Figure 3.1: grind Toolflow

While the source code could go through faulty compilers before reaching the intermediate software (that will be compiled into the hardware representation), previous modern techniques have been introduced to ensure software compiler correctness. Relying on these works and assuming that the software intermediate representation of the code is accurately representative of the source code, we consider this high-level specification as a golden reference for the behavior of the circuit. This method, used for example in [19, 18], is often referred to as Discrepancy Analysis [17, 56].

On the intermediate software representation side, our goal is to extract enough information to A) correctly map back the hardware components to their corresponding software presentation B) anticipate the correct set of outputs for each component and C) extract the dependence graph and node information to be further used in our verifying technique.

This is done by instrumenting the LLVM representation, two JSON files are generated, one containing the golden values which is the set of IDs and the list of their outputs, and the other containing the dependence graph and node information. Code 3.1 shows a few lines in the extracted file dependence graph file. The information is that An adder with the ID of 22 and the parent basic block with the ID 19 has three parents (predecessor) nodes with the IDs $20, 21, 22$. The steps to generate this file are listed in the Appendix.

The golden values collected from the LLVM can be used as *patch values* in the verifier. The guard function can be manipulated to perform many things. However in its default setting (which is for the verifier) the guard function will report an error if a node's output does not match these golden values, but more notably, the guard function will "patch" the faulty output with the correct golden value to isolate the fault, hence, the name patch value. The advantages of isolating the fault are further discussed in the next chapter.

```
"id" : 22,
"instruction" : "  %add = add i32 %mul, %0, !dbg !63, !UID !65",
"name" : "binaryOp_add22",
"operands" :
["INS_13", "INS_21"],
"parent_bb" : 19,
"type" : "Binary"
"parent_info" :
```

```
[ 20, 21, 22]
```

Code 3.1: Line 273-282 in Relu.muir.JSON which contains the dependence graph information of the Relu benchmark. The information in this code is available for all node IDs

3.2 Auto-Wiring guards into H-RTL

Figure 3.1 illustrates the passes we have developed to attach guards to the H-RTL. The example illustrates a simple address verifier that analyzes the loads in the H-RTL circuit. As mentioned in the previous chapter, we gather data flow information from the LLVM representation. In this Metadata, the IDs of nodes are also gathered, an ID is a unique decimal number assigned to each node and its corresponding hardware component. Keeping this mapping information also allows the user to indicate their region of interest at the program level and we can track down those signals. In step 2 of Figure 3.1 the guard list is filled based on the instrumentation goals e.g., load nodes (The iterative verifier introduced in the next chapter will generate this set of nodes automatically). In step 3, grind iterates over the H-RTL and identifies the signals (registers and wires) within the module. For each signal, grind attaches a guard in the H-RTL module. In this example, since loads are instrumented, the address and data fields are annotated. In step 4, we define the guard circuits and connect them to the actual signals.

Before we go into the design of guards, one major challenge to answer is: *How to connect the guards to the main circuit's components and remove them automatically?* In IR connections are hardwired using input and output ports and handshaking signals, however, that is not suitable for *automatic and conditional* connections.

grind leverages FIRRTL, a compiler that loads H-RTL into a data structure that we can transform and rewrite. The main challenge is guards are separate modules introduced post-H-RTL generation, while the module signals could be embedded deep in the H-RTL's module hierarchy. To wire these up grind uses a FIRRTL pass that "bore" through the module hierarchy (`https://bit.ly/3ycg5aQ`; Figure 3.2 illustrates. The actual bore implementation is included in Appendix A. A bore connection consists of a sink and a source, each defined in a separate class in Chisel, and connected through a unique "string" input. To mimic handshaking connections with Bore, we create three bore definitions for each conditional attachment: ready, valid, and data. The main difference between the bore handshaking connections and the input/output based connections is that the bore connections can be placed inside an "if" statement, making their existence conditional to a single enable button, and easy to remove. Code 3.2 Shows an example of Boring connections used in grind, this particular code is added to the actual module that the guards are being attached to. A sink connection acts as the receiver side and a source connection as the source. The ID of the module creates a unique string on the source side of the Bore connection that will be connected to the Sink side of the connection, which is in another module with the same unique string.

```
//import Boring
import chisel3.util.experimental.BoringUtils

if (Guard_enabled) {
        //Input data from guard
        BoringUtils.addSink(in_data, s"in_data${ID}")
        BoringUtils.addSink(in_value_valid, s"in_valid${ID}")
        BoringUtils.addSource(in_value_ready, s"in_ready${ID}")
}
```

Code 3.2: Boring connections in a module code, the other sides of these connections are guards. The if statement creates a condition upon the existence of the guard's attachment to this module.

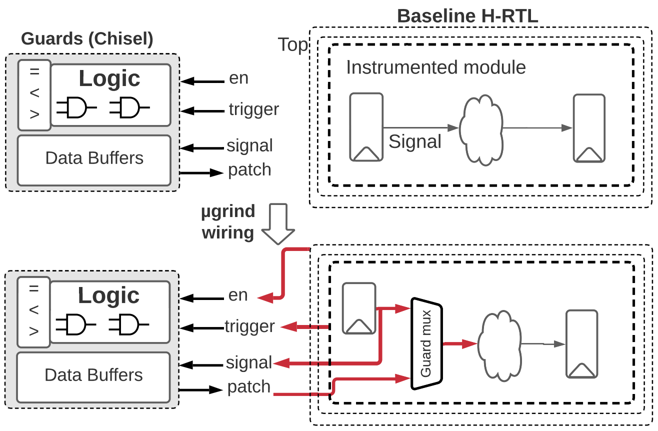


Figure 3.2: Boring wires between guards and nested H-RTL

3.3 Guard design

Each guard monitors an H-RTL signal and includes five components: i) **Trigger:** a boolean enabler signal that activates the guard. ii) **Reader buffer (shadow RAM)** The metadata is streamed from DRAM during the execution. Since a node can execute multiple times it's buffered to the guard, since each module can be executed many times the buffers hold the values for each execution. iii) **Guard function** a logic block that uses the incoming H-RTL signals and input buffered values to calculate a patch value. The majority of analysis functions require simple logic, e.g., isEqual() or isRange() that can be accomplished in 1 cycle. iv) **Patch value:** The patch value overwrites the H-RTL signal during execution. Patches are useful for patching erring signals during debugging and injecting faults for testing resiliency. They can be used to fix or break the circuit v) **Writer buffer**. Each entry includes: i) runtime context: logical timestamp and cycle time when the guard was triggered. ii) the signal values from the H-RTL, and iii) the output of the analysis. The guard function and the Reader buffer are shown in Figure 3.3 as the Analysis process to patch the faulty output of Load A[i], and the writer buffers are shown outside the guards transferring the guard data to the memory. The two sided connections between the guards and the modules are implemented using Bore connections.

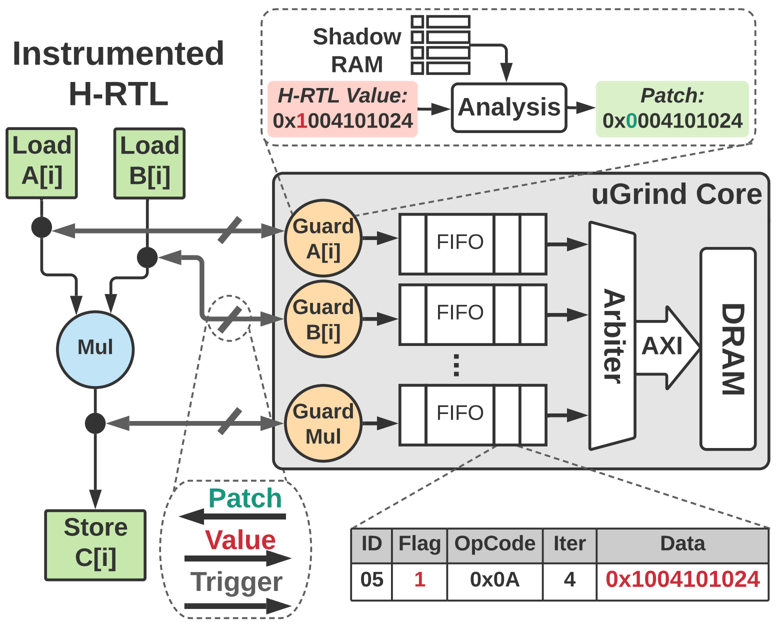


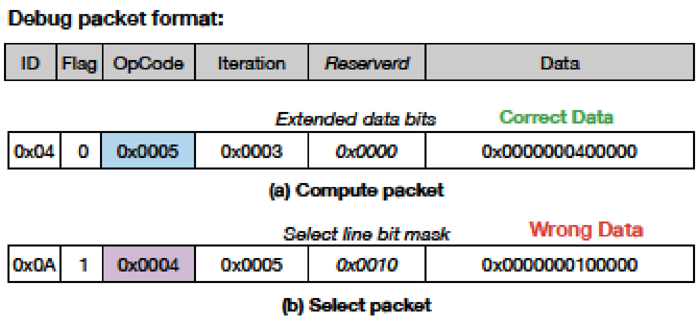
Figure 3.3: Guards wrapped indie the circuit

These traces are collected in the form of packets and must contain enough information to allow grind to further analyze and give the user a valid insight into what has taken place in the module, to help the user determine the possible root cause of the fault and how to fix it. As mentioned previously, each node in the design contains a unique ID, and an opcode that gives information about the functionality of the node. There can be nodes in the design that have multiple functionalities and at each iteration, they may execute one of those functions. There is also an iteration counter that helps to keep track of the partial ordering of the packets. There is a flag that indicates whether the node has generated a valid output or not, and a reserved field in the debug packet that can be used for embedding more information during the execution. Figure 3.4 shows two examples of such packets. In this example, the first packet (a) is been generated by node number 4, the node is a compute node, OpCode=0x005, and the node the output data is correct. However, the second packet (b) is been generated by node number 10, the node is a select node, OpCode=0x004 and the data is wrong. Finally, when the execution of the accelerator finishes, the writer buffer dumps the data into the memory. The structure of the recorded packet can be updated based on user demands.

There are four files per each guard. The Reader and Writer files that implement the buffering of data and the hard-coded connections to the memory, parts of the Reader file are shown in Code 3.3. And Code 3.4 shows an example of a guard function implementation, in this scenario the implementation is the default (verifier) implementation where the guard patches the node's output. The Boring connections connecting the guards to the actual module are not implemented in this file.

```

val Data = Module(new Queue(UInt((xlen).W), BufferLen))
val queue_count = RegInit(0.U(log2Up(BufferLen).W))
```


Debug packet format:

ID	Flag	OpCode	Iteration	*Reserverd*	Data

				Extended data bits	Correct Data
0x04	0	0x0005	0x0003	*0x0000*	0x0000000400000

(a) Compute packet

				Select line bit mask	Wrong Data
0x0A	1	0x0004	0x0005	*0x0010*	0x0000000100000

(b) Select packet

Figure 3.4: Data Recorded for Each Node

```
when(io.vmeOut.data.fire) {
    queue_count := queue_count + 1.U
}
            ...
Data.io.enq <> io.vmeOut.data
            ...
switch(rState) {
    is(sIdel) {
        when(Data.io.count === 0.U && io.out.ready){
            rState := sReq
        }
    }
    s(sReq) {
        when(io.vmeOut.cmd.fire()) {
             rState := sBusy
        }
    }
    is(sBusy) {
           when(queue_count === (BufferLen - 1).U) {
             rState := sIdel
             addr_reg := addr_reg + (queue_count * (xlen >> 3).asUInt())
           }
    }
}
```

Code 3.3: The implementation of the inner part of the guards, here, the Reader buffers the data from the memory.

```
if (Guard_enabled) {
    when(FU.io.out =/= golden_value) {
        isBuggy := true.B
        //correct the output
        io.Out.foreach(_.bits := DataBundle(patch_values.get(guard_index),
    taskID, predicate))
        }
            ...
}
else {
    io.Out.foreach(_.bits := DataBundle(FU.io.out, taskID, predicate))
}
io.Out.foreach(_.valid := true.B)
ValidOut()
```

Code 3.4: The implementation of the guard function for the verifier. In this instance the guard function compares the node's output to the patch value. If they don't match the isBuggy flag is set to true and the output is patched (corrected) by the guard, In the verifier, patch values are equal to golden values.

3.4 Guard wrappers

We know so far that each guarded component will have an attached guard, and this guard contains at least an input buffer, writer buffer, and a guard function. The connections between the guards and their guarded components or memory are fully 'bore'd and conditional. In each execution, there will be multiple guarded components. Guard wrappers serve as the top module for all the guards mixed in with the H-RTL (Figure 3.3). Having a separate guard wrapper can provide shared I/O for the user to access the guards. If guards were implemented as part of the H-RTL modules, then the I/O ports of H-RTL modules would have to be redefined. guards can be manipulated to exchange information with each other for dynamic analysis. The writer wrapper collects the results of the analysis and writes them to memory using buffers. An important issue we had to consider was how to handle the write buffers filling up. We keep the circuit completely decoupled from the H-RTL and drop packets if the buffers fill up. Note that in this case, the guards themselves continue to function, analyze, and patch values if required. We only drop the outputs for some cycles. However, this approach continues to maintain the timing independence and fidelity of the H-RTL circuit. Code 3.5 shows parts of the Reader Wrapper code. "numGuard" is the number of the nodes that are being guarded. And "boreIDsList(i)" is the list of the IDs of the nodes that are being guarded. For each guard connection a unique string is created to connect the Reader buffer for that node to the node itself. The same implementation is done for the Writer buffer and wrappers.

```

val read_buffers = Seq.tabulate(numGuard) {i => Module(new ReaderBufferNode(ID =
     i))}
val read_buf_data = List.fill(numGuard)(Wire(UInt(xlen.W)))
        ...
for(i <- 0 until numGuard){
    read_buffers(i).io.addrguard := io.addrguard(i)
    io.vmeIn(i) <> read_buffers(i).io.vmeOut

    read_buf_data(i) := read_buffers(i).io.out.bits
    read_buf_valid(i) := read_buffers(i).io.out.valid
    read_buffers(i).io.out.ready := read_buf_ready(i)
    //Boring connections to the node
    BoringUtils.addSource(in_data(i), s"in_data${boreIDsList(i)}")
    BoringUtils.addSource(read_buf_valid(i), s"in_valid${boreIDsList(i)}")
    BoringUtils.addSink(read_buf_ready(i), s"in_ready${boreIDsList(i)}")
}
```

Code 3.5: The implementation of the Reader wrapper.

The designer can direct grind to remove the guards from the finalized H-RTL this can also happen automatically. We use FIRRTL to manipulate the H-RTL and remove the guards [1]. As shown in Figure 3.1 grind includes two interfaces for the designer to specify which guards to build in, i)a global table and ii) a per-module flag (e.g., shown here `https://bit.ly/3j21U3k`). To remove all guards the designer has only had to unset a flag in a JSON file for that ID. grind will not include any guard annotations and FIRRTL will cut the wiring as shown in Figure 3.2. This makes the guard I/O ports dead i.e., isolated logic with floating I/O ports. Subsequently, dead-code elimination `https://bit.ly/2WxYXyj` will mark and remove the guards.

Chapter 4

Verifier Tool

In this chapter, we develop a grind tool that leverages guards to verify the correctness of each component in a circuit. The Verifier validates the HLS translation from C-to-hardware and performs equivalence checks fully automated and without any user intervention (apart from the initial interaction). grind Verifier includes the following novelties:

- We pinpoint the exact cycle and additional information an H-RTL signal deviated from the software values, this additional information can be adjusted as shown in Figure 3.4. As far as we are aware, we are the first to demonstrate error checking of end-to-end accelerators.
- We create an iterative approach that retains only the scope of an erring H-RTL signal in each iteration. This helps check larger H-RTL circuits with less memory usage.
- The H-RTL circuit can continue execution even if an error causes a signal to deviate since guards patch correct values during execution. This enables us to catch **multiple errors and prevent hard-stop failures**.

As we go through the behavior of each node in hardware, we look for the nodes that show unexpected behavior in comparison to the software Metadata (golden values). This misbehavior can be detected by placing a comparator on each one of the output values or signals (that have a corresponding golden value). For this framework and the limitations on space and performance trade-off, we place all guards on the output values of modules (nodes). This means that the guard function will receive the golden values as an input to compare against the node outputs and patch the node outputs with the golden values if they don't match. Nevertheless, data dependency can be the root cause of many reported bugs, a faulty segment of the design can propagate faulty values through every path that is data dependent upon it and introduce several correctly implemented components as buggy. merely because they were fed the incorrect data propagating through the dataflow. This will make finding the root cause of incorrect behavior more difficult for the user and doesn't allow the user to catch other faults unless they fully fix the first one that showed up. And even worse, it can sometimes interrupt execution.

In [28] the corresponding signals that are needed to determine the dependencies have to be recorded. The trace data is then analyzed to determine which conditional dependency was relevant for the captured execution. However, memory space on chip is limited. And recording all the needed signals to map back dependency bugs to their root causes would require memory space and human post-processing. In this work, we take on an approach to isolate faulty locations from the rest of the circuit and make sure that these bugs do not propagate through the execution. We do that by implementing the guard function as a comparator that corrects the output of a node. Verifier guards use unique IDs to map the hardware modules to their software representation, they receive the values extracted from the software intermediate representation (golden values) for each node alongside their IDs, check the hardware output with the same ID against them, and rewrite the output if they did not match. Hence, isolating the rest of the circuit from this misbehavior.

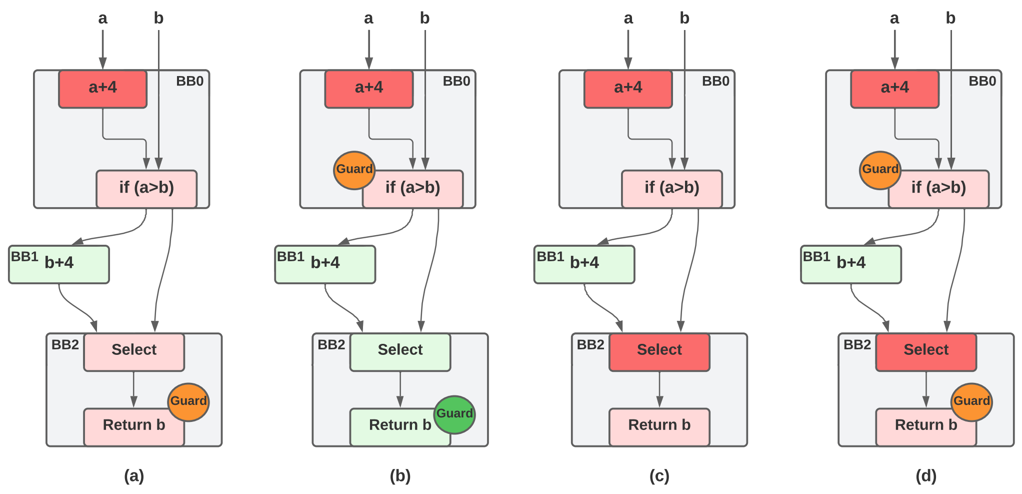


Figure 4.1: Guarding the right location isolates the faulty area: The a+4 adder is producing an incorrect result. a) The Return node is guarded and the fault propagates through the circuit. b) the if statement is guarded and the fault remains inside BB0. c) both *a*   4 and *select* are faulty. d) the outputs of BB0 and BB2 are guarded.

(a) Flagged guards	(b) Flagged guards	(c) Flagged guards	(d) Flagged guards
Return b	if(a>b)	-	if(a>b), Return b

Table 4.1: Statements that are reported as faulty in each Figure 4.1 case.

To further look into the usage of guards in dependency bugs, let us look into the dataflow shown in Figure 4.1, a Control-Flow Dependence is shown where the result of the add operation *a*   *a*   4 puts a condition on the correctness of the path the program will take as well as the computational results. In Figure 4.1 (a) and (b) this adder (*a*   *a*   4) misbehaves, let's say that it decreases *a* by the amount of 4 instead of increasing. The if-else statement will act incorrectly as well, and *b*   0 is returned, which causes the return node output to be incorrect. Now let's assume that the return node is guarded and its output is corrected, and a fault flag is set to true. as Shown in Figure 4.1(a), (green stands for module correctness, dark red for root fault, and light red for propagated fault). However, the bug was initiated in a 4 if we guard the $if\ a\quad b$ statement, this bug will not propagate past that location and will stay within the BB0 block, we call this process isolating the fault. Now let's say there are multiple bugs in the circuit as shown in Figure 4.1(c). in this case both a 4 and *select* are faulty, meaning that the select node will both get a wrong input and is faulty itself. Two things can happen: i) the select node produces a wrong output as well. ii) the two faults cancel each other out. and the select node (even though faulty) will produce a correct result. The way to effectively capture both faults in this scenario is to guard both output statements of each basic block BB0 and BB2 (We refer to the outputs of basic blocks as Live-outs). Figure 4.1(d). Here, both *Returnb* and $if\ a\quad b$ are flagged as faulty and even though both of them are faulty because of their parent nodes a 4 and $select$, the faults are isolated and by guarding the parents the user will capture both a 4 and $select$. In a trace-based approach, the user first had to capture a 4 as the first location of divergence, fixed it and then figured that *select* is faulty too in the next runs. Table 4.1 further shows what the user is seeing in the guard reports in each case.

4.1 The iterative Verifier

In previous methods where a trace of the hardware execution is recorded, either a user or a post-processor has to go through the signal and value trace to look for the place the first divergence happened. Moreover, in this scenario, since the faulty component was not isolated from the rest of the dataflow the faulty value will propagate and taint the rest of the path, so the user has to fix this module first, and repeat the process to find further buggy behaviors in the rest of the circuit, moreover, in larger circuits, recording every signal and value will become a huge memory overhead, pushing designers to choose between recording parts of the needed information for optimization purposes or record everything in multiple runs. Our proposed debugging algorithm improves upon classic bug-recording techniques and takes advantage of the guarding interface alongside the Metadata extracted from the intermediate software. We aim to record every place that the hardware diverged from the correct path (golden model) while dismissing divergence caused by dependence (this is where the dependence graph information collected as Metadata comes to use).

Figure 4.1 gave us an idea of how isolating faults can help capture them more efficiently. One way to use guards to isolate the faulty components from each other and the rest of the circuit is to set guards and every possible component's output. Doing so, in one run, every node behaving functionally incorrect will trigger its guard to replace the output and set the faulty flag to true, presenting the user with a list of the buggy components. With this knowledge, the user will have a clear notion of the buggy components in the circuit and will further have to examine those components to understand the root cause for that component's buggy behavior. However, guards come with a cost. for each guarded node, a guard function and read/write buffers are placed in the circuit. This amount of added computation and memory usage will have an overhead on the limited memory, or on the timing and performance overhead of simulation. Keep in mind that usually, only a small and limited part of a circuit is buggy, so this approach will practically waste memory and time on processing

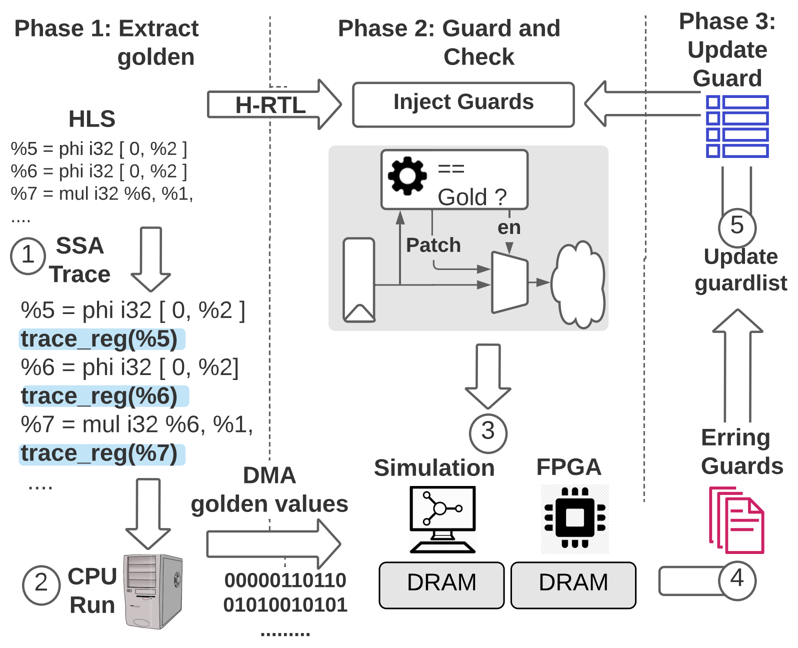


Figure 4.2: The grind iterative H-RTL verifier toolflow.

correct units and fall into the same problems that tracing techniques face. We present an effective algorithm that takes advantage of guards and the golden values while optimizing memory usage.

Algorithm 1 and Figure 4.2 illustrate this Verifier. As shown in Figure 4.2 the grind works in three phases. Phase 1:Extract golden is run once on the LLVM code, data is stored to a .JSON file. 1 The HLS compiler [54] dumps the program-dependence graph and the golden values. 2 We insert software instrumentation to trace the SSA registers and memory live-outs from a CPU run. The golden values are written to the DRAM, and are consumed by guards during the FPGA or simulator run. Phase 2: Checking. We instrument the H-RTL signals of the guarded modules (the next section discussed how we choose those modules). In 3 , the instrumented H-RTL is synthesized and mapped onto the FPGA. In 4 , the guards write the IDs and information of the mismatched signals to the DRAM. In Phase 3: Update guards, we use the guard output to identify H-RTL signals that deviated from the SSA registers. 5 We then backward slice the SSA form and update the guard list to include the control/data predecessors i.e., we check if the errors originated earlier in the circuit, and with guarding each slice we isolate the positives. We keep iterating phases 2 and 3 until we find the actual faults. The actual implementation of this tool is further discussed in the Appendix.

Algorithm 1 shows what happens during the Verifier code execution at a higher level. First, we need an initial set of nodes to inspect (since we don't want to guard the entire circuit) we define the initial set of nodes as Memory ops, control ops, and live out values, since any Basic Block's output will show in one of these forms. So basically, we are setting inspecting units at the end of each basic block. If there is a faulty component inside that basic block, it will show up in one of these forms and then the Verifier knows which basic blocks contain faulty components and focuses on them. Step 1: we insert the guards to the initial list and in Step 2 we run. After this run, we know exactly which Memory ops, control ops, or Live-outs were faulty, keep in mind that these faults could either be because they were faulty, or because they were fed faulty values by their predecessors. This is where the dependence graph in the collected Metadata comes to use. We use the dependence graph to find the predecessors of these faulty initial nodes and update the list of the next iteration nodes to i)the nodes that showed a fault in the previous run ii) their predecessors in that basic block. If there is not enough memory space for all their predecessors, we can add the predecessors in smaller amounts, even one by one in each iteration. and this is step 3, updating the guards. After iterating on the entire set of faulty predecessors the verifier is done, if a node had produced a faulty value at first but generated a correct value when its predecessors were guarded the node is declared as not faulty, and if it still produced the wrong value after receiving the correct guarded inputs, it is in fact faulty. The implementation of the verifier tool is further explained in the Appendix chapter.

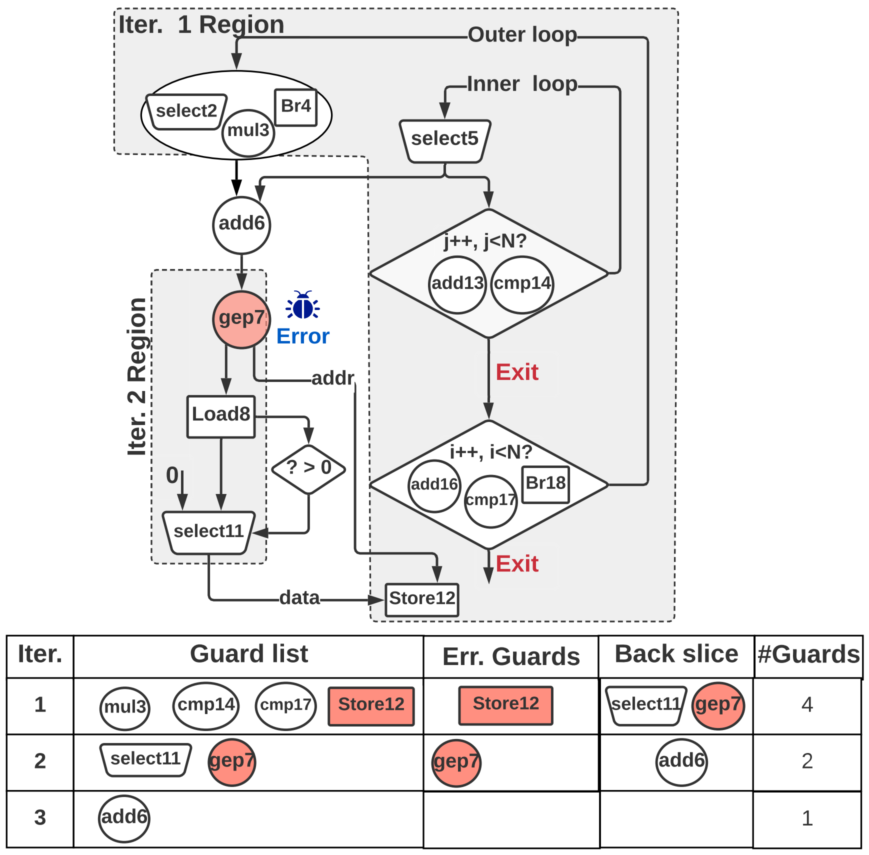
Algorithm 1: H-RTL Verifier

```
   Global: Guards = [Memory and Control ops, live-out components]; ROI =
1  Verifier(Circuit) while Guards[] predecessors are buggy do
       // Inserts guards
2      foreach g in Guard[] do
3          Circuit.add(g)
       // Run circuit with guards, simulation or on FPGA
4      ROI = Run(Bit, Golden[])
       // Backslicing, add guards to the predecessors of buggy components and
          remove guards from correct components
5      foreach signal in ROI do
6          if signal.failed() then
7              Guards.add(signal.predecessors())
8          else
9              Guards.remove(signal)
```

4.2 Working Example

In this section, we illustrate the tool with an example. Figure 4.3 shows the Relu data flow, a nested loop. The numbers in each dataflow node indicate the SSA register id. in this example the gep7 node (7 is the node ID) has a stuck-at-error that causes the faulty value to propagate and taint its successors e.g., $load8$ $select11$ $comp$ $store12$. The goal of the checker is to report only gep7 to the user, while if we had run this circuit with a tracer, $load8$ $select11$ $comp$ $store12$ $gep7$ would have mismatched from the golden value. And aside from the large memory trace, if let's say select11 was also faulty, the user would have no way of knowing. The Verifier must first observe Store12 as faulty in its initial run, and then zoom on only Store12's predecessors (that includes gep7) and at the end, present gep7's ID, the cycle on which it produced the fault and the faulty data to the user.

Going through the steps that the Algorithm will take to identify the fault: In the first iteration, the guard list is initialized to the live-outs, control-nodes, and memory nodes in the H-RTL. The guards will compare the outputs of these nodes to the golden values and report store12 as faulty, meaning the data packet for ID 12 will have the fault flag set to 1 and Guards will patch the output of store12



Iter.	Guard list	Err. Guards	Back slice	#Guards
1	mul3 cmp14 cmp17 Store12	Store12	select11 gep7	4
2	select11 gep7	gep7	add6	2
3	add6			1

Figure 4.3: Illustrating each phase of the H-RTL verifier in a Relu circuit. gep7 has a stuck-at-zero error.

with golden values of the to ensure subsequent nodes (if available) are not tainted. The intuition here is that guarding these nodes in the datapath help 'narrow' the region of interest, since we no longer will look into nodes such as add13, and cmp14, they are irrelevant. In the second iteration, we guard the backward slice of the store12 using the dependence graph information, select11 and gep7 are guarded. In this iteration, gep7 will fail the checks, but store12 will work correctly as gep7 is guarded and patched now, load8 and store12 will be getting correct inputs. In the final iteration, we guard gep7's backward slice, its predecessor add6. Since add6 is not faulty it will not set the fault flag to 1 and we find the true fault i.e., the failing guard from the previous iteration, gep7. The efficacy of the Verifier is determined by the successive refining and trimming of the guards in each iteration. As long as the memory space allows us to guard the initial list, the Verifier will manage to verify the circuit in a number of iterations based on the limit.

Chapter 5

Additional Tools

In this chapter, we look into two additional grind tools to demonstrate the flexibility of guards that provides a wide range of tools that can be defined based on user demands. As we previously mentioned, guards are instrumentation tools and can be manipulated to perform multiple tasks. While the grind verifier is the main tool we demonstrate in this thesis, the overall idea behind guards is simple: functional units of hardware that are easily attached to and removed from different components of the circuit, to inject, observe, manipulate, or process data; or any combination of these tasks. the guards can be manipulated to implement any data processing unit on the signals of the component they're attached to, with or without outsourced data.

The two other tools that are introduced in this chapter are as follows:

- **H-RTL Profiler** A smart guard-based profiler that dynamically tracks, processes, and records based on the user's needs and demands.
- **H-RTL Faulty** A fault injector, a tool for purposefully breaking the circuit and injecting faulty values in the circuit. This tool can be used to either inject faults to test certain debuggers, or used alongside the profiler to inspect the effects of a faulty module on the behavior of the circuit.

5.1 H-RTL Profiler

In this section, we build a smart dynamic profiler for H-RTL circuits. When a designer looks into the traces of a circuit execution, they usually only seek certain values and data which they grep out of the large full trace. And again, they usually will have to perform a set of logical operations on that data to finally get the results they wanted to look into. Prior state-of-the-art [33] has relied on out-of-circuit performance counters. This leads to high DRAM traffic, wastes on-chip SRAM, and slows down RTL simulation.

Since profilers tend to be relatively simpler circuits, consisting of a memory operation to record the data and a simple logical unit to process the data beforehand, embedding profiler circuits in the H-RTL and profiling dynamically during the execution is a better alternative to extracting H-

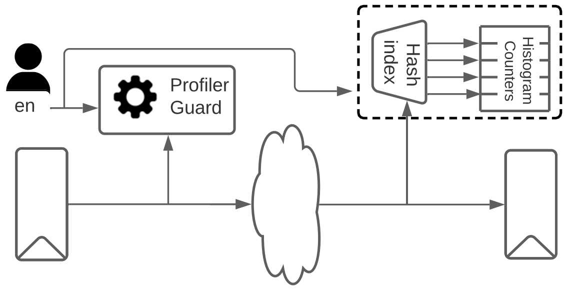


Figure 5.1: Using guards to profile H-RTL signals.

RTL signals for post-processing, as opposed to using a large section of memory to trace everything and later have the user go through the trace. Figure 5.1 illustrates our approach. Profiler guards are lightweight circuits: i) **Read-only** the profiler guards only read the signals and do not patch in any values; this eliminates expensive muxes. ii) **Auto-enabled:** the profiler are automatically inserted into the H-RTL and can be dynamically enabled. This enables the profiling to be tuned. iii) **On-chip Stats:** Finally, the user can flexibly track the statistics on-chip. An example of this sort of profiling can be activity factor profilers, let's say that the user wants to know how many memory accesses were made during a time cycle interval (a,b). In a trace-based profiler, post-processing of all the cycle activities has to be made to find this number. However, in the grind profiler, counter guards will be attached to load and store components, check the cycle each time these components were active, and increment the counter for the ones that were active in these cycles.

5.2 Guard based fault injector

Next, we study H-RTL circuit resiliency using guards. We use two types of guards for this dynamic instrumentation. As you might recall, we previously used guards to fix faulty modules, meaning if a module produced a wrong output the guard fixed it. Here, we do the exact opposite, a fault injector guard will toggle the output of a node to behave faulty. We will also again use a certain type of guard that like the verifier guards compares the outputs of the other nodes in the datapath to their golden value and reports the mismatch, but doesn't correct them. This way we basically break a part of the circuit and observe how it affects the rest of the circuit. Figure 5.2 shows the guards that were injected. We name the two types of guards that are used as faulty and checker. Faulty purposefully injects buggy values into the circuit signals when enabled to break that module. Checker guards check the values against golden values and report if they deviate without patching. This allows faults to propagate through the circuit and affect different parts of the circuit, a designer then can learn how different faults affect the circuit and evaluate how different faults can cause different types of failures. Using this tool the HLS compiler designer gains a better understanding of circuit resiliency and devises circuit transformations to guard against them. The three categories of bugs we inject are i) **Compute Fault:** We flip the bits and inject stuck-at-zeros in the ALUs ii) **Control Fault**

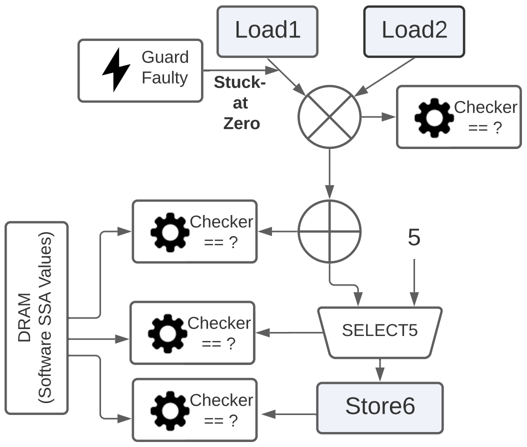


Figure 5.2: Example of Using guards to inject faults into Load,1 nodes while verifying resiliency in other nodes in the path.

In this case we introduce faults in the branch and merge operators ii) This can result in incorrect operations executing in the H-RTL circuit and even deadlock. iii) **Memory Fault:** Finally we pert rube the memory addresses of memory operations and check the impact on the final memory state.

Chapter 6

Evaluation

In this chapter, we study the functionality and performance of the grind iterative verifier, H-RTL profiler, and H-RTL faulty. We have picked *Reli Saxpy FFT Gemm Conv2D*, and *Stencil* applications, source: Machsuite [50] benchmarks. We choose these designs because they have different design elements, which shows that grind can be used to verify different designs. The applications like FFT, Gemm, Conv2D are floating-point applications, and Saxpy and Relu are workloads that have high bandwidth. The complexity of these benchmarks is discussed in Table 2.1.

6.1 grind verifier evaluation

We introduce two specific case studies and discuss the framework in practice. Using injected faults and a real example we show how grind Verifier iteratively zooms in on the faulty components in the accelerator. And using synthetic injected faults we study the performance of the tool.

6.1.1 Case Study I: Analysis of the Verifier by injecting faults

To evaluate the Verifier we study two forms of deployment: running co-simulation within a Verilog simulator and the other a deployment on Amazon AWS F1, Xilinx UltraScale+ FPGAs. Our evaluation criteria are i) **SRAM usage and bandwidth** (§ 6.1.3): The resources consumed by the guards impact the H-RTL size we can support and large circuits set limits on the grind guards. ii) **Time-to-check**: This corresponds to the time taken to identify the faults or confirm the lack of them (under the specific tests). It depends on the number of iterations required and whether we have to rebuild the guard list onto the FPGA. iii) **H-RTL size:** We measure the size of the design-under-test as a % of the total number of logic resources available on the FPGA. Higher % implies we can check larger practical designs, without having to scale them down for prototyping and instrumentation.

Observed results:

The grind verifier successfully captures multiple bugs in the same amount of iterations as a single bug, given unlimited memory.

The grind verifier requires 2—10 less on-chip SRAM than state-of-the-art trace-based checkers [56]. grind's iterative approach only checks a subset of signals in each iteration.

The grind verifier can verify circuits 2—5 larger compared to prior work. We can progressively scale up design size as we narrow the guarded site.

The grind verifier can rapidly trim the circuit to discard correct parts. Guards on nodes that show no sign of fault will be eliminated by our backward slice in each iteration, which means less than 3-4 iterations required to check a circuit with unlimited memory.

All accelerator circuits are verified end-to-end in 24hs, if we run them on FPGAs. And less than 2 hours in simulation. Prior state-of-the-art expect humans to manually identify fault from trace, a process that could take weeks.

6.1.2 SRAM usage for the grind verifier

The average memory usage of the verifier is dependent on the size of the design, test input size, and the location of the fault in the data flow. We observe that as long as the fault is impacting the functionality of a module inside the circuit the grind verifier will capture it. We introduced multiple arbitrary bugs in our benchmarks and in all the cases the verifier correctly pinpointed the buggy location. We further realized that even though the initial memory usage for all bug locations is the same since the initial list is constant ($Live\quad outs\ controlandmemoryops$), the amount of memory that guards consume reduces significantly in the next iterations if the bug was placed in the outer loops. In this chapter, we refer to bugs that are embedded deep in the inner loops as bugs in "hotspots". Table 6.1 shows the memory usage per iteration for the grind verifier to capture multiple bugs in hotspot locations of the benchmarks. As we can observe, with no limit on memory, the verifier captures all the bugs in 3 iterations since, with no limit, all parent nodes of a faulty component inside the basic block can be added to the guard list in the following iteration. As we can see in Table 6.1 the second iteration uses more memory than the first iteration, that is because since the bug is in the most inner loop (hotspot) after the initial list report most of the nodes in that region will be guarded.

Relu (KB)	193.1	320.0	64.5
Conv2D (KB)	889.0	1842.6	182.2
Saxpy (KB)	23.0	54.1	7.5
Stencil (KB)	516.0	580.8	82.0
Gemm (KB)	417.0	1536.4	128.7

Table 6.1: Memory usage for hotspot bugs.

Next, we look into the effect of injecting bugs in different layers of the loops. We expect the bugs in the most inner loops (hotspots) to have the highest average memory usage and the bugs in the most outer layers the least memory usage. We use the Conv2D benchmark to illustrate this effect since it has a 4 level nested loop. Table 6.2 shows how the bug placement affects the memory usage, as we see when the bug is located in the outer loop layer the memory usage decreases significantly in the 2nd and 3rd iteration.

Layer1 (KB)	889.0	0.56	0.09
Layer2 (KB)	889.0	2.2	2.2
Layer3 (KB)	889.0	141.6	20.2
Layer4 (KB)	889.0	1842.6	182.2

Table 6.2: Memory usage for Conv2D based on bug location. Layer 1 means the most outer layer of the loop and layer 4 is the most inner layer.

Finally, we will look into a case where memory is limited. We use the Relu benchmark with a larger input size and a memory limit of 600KB. In this case, the memory limit will not allow the verifier to add all the faulty node's parents to the guard list immediately after the fault flag is set to true, and must add them separately which will increase the iteration count. We insert the bugs in 4 different locations (*adder gep select mul*) and report how many iterations it took for the verifier to finish for each bug. The results can be seen in Table 6.3, with unlimited memory it took 3 iterations for all bugs.

Bug1 (adder)	12
Bug2 (gep)	9
Bug3 (select)	11
Bug4 (mul)	4

Table 6.3: Number of iterations the verifier took for each bug in Relu with the memory limit of 600KB.

6.1.3 SRAM: State-of-the-Art vs grind

We compare the amount of SRAM required against Autoslide [56] the state-of-the-art trace-based checker. To obtain a complete picture we study errors from both hotspot regions and low activity spots. Overall (Figure 6.1), grind has a lower SRAM requirement relative to AutoSlide, while maintaining a low time-to-completion. grind requires 2–10 less SRAM than trace-based approaches, as expected since we start with a limited set of components and trim down on them. Also, traces detect the bug offline and need to collect the region of interest. Since there is no oracle, traces tend to be collected in a coarse-grained manner across a large portion of the circuit (including those functioning correctly). grind spreads the verification over multiple iterations starting with fewer components. In each iteration, we dismiss the correctly functioning parts of the circuit. The amount of SRAM required is proportionate only to the activity factor and % of the circuit tainted by the faulty signals. When the instrumented circuit is lowered onto an FPGA there is a trade-off between time-to-check and H-RTL size. The verifier supports different flows, here we look at three(Figure 6.12):

- **1-synthesis**: We synthesize the H-RTL once onto the FPGA with the complete guard set. All guards are built into hardware at the beginning, limiting the logic left-over and consequently H-RTL size.

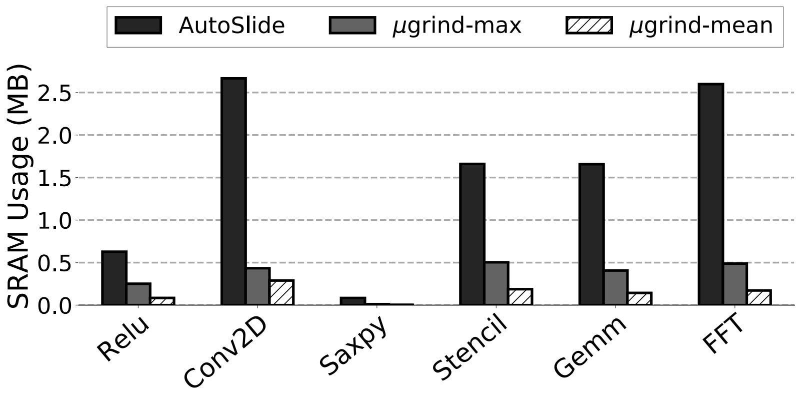


Figure 6.1: SRAM requirement of grind vs Autoslide.

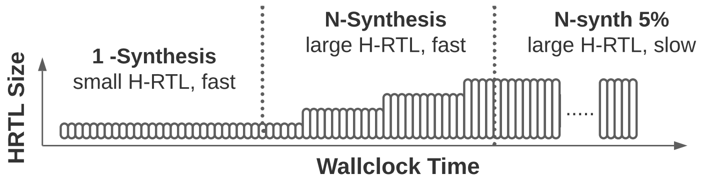


Figure 6.12: Tradeoffs between different flows in H-RTL Verifier.

- **1-synthesis**: grind only builds in the guards in each iteration. As the guard list trims down, the amount of used memory lowers as well. We observe that the highest memory usage happens in either iteration 1 or 2. In this scenario we can also bound the guards to K% of LUT and BRAM, leaving 100-K% of the FPGA for the design. This limits the number of guards per iteration, and we require more iterations to check the required nodes, referred to as **N-synthesis K%** in Figure 6.12 with K set to 5.

Something else we can set a limit on is the bandwidth. The bandwidth required by the verifier depends on the number of guards in each iteration. We observe that with the injected bugs the bandwidth usage increases between 2% and 7% for each benchmark.

We introduce two metrics i) $\frac{\#Guarded}{\#Bugs}$: The ratio of the number of netlist signals guarded to the number of actual faults in the circuit. The number of bugs is fixed, which means this effectively measures the wasted guards in each iteration. ii) Accuracy: The percentage of signals and registers guarded in the overall circuit. This indicates the overhead of guarding in each iteration. Figure 6.15 shows how we improve accuracy as we converge on the bugs. We highlight the rate of change i.e., the higher rate implies that grind zooms in on the bug faster. Figure 6.14 shows the percentage of the circuit that was guarded in each iteration. The lower the % lower the overhead. We find that on average 69% of the guarded H-RTL signals will be trimmed in each iteration by the backslice. Note that this rate is not fixed and varies across iterations and benchmarks. It depends on both the H-RTL dataflow, actual dependencies, and site of error. These factors are considered in the average.
Finally, Table 6.4 summarizes the benefits of grind in comparison to prior work.

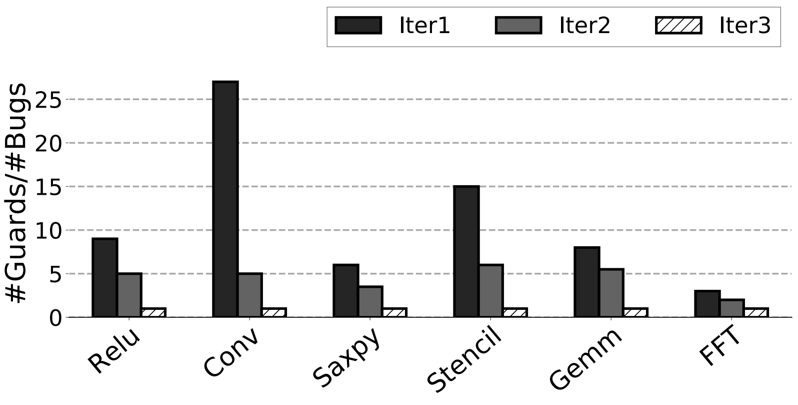


Figure 6.14: $\frac{\#Guards}{\#Bugs}$ rate in each iteration.

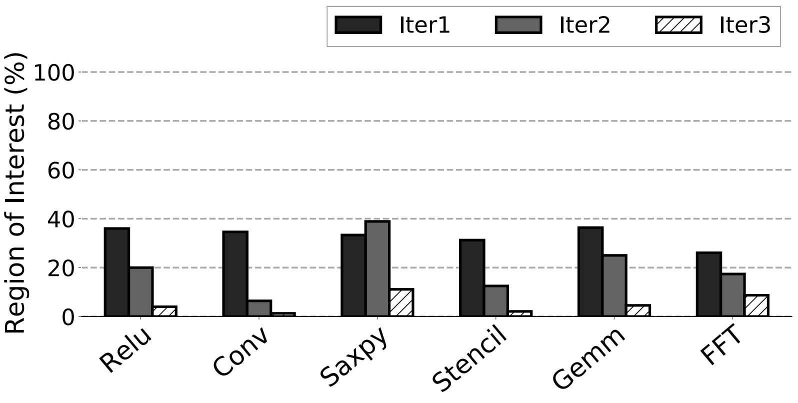


Figure 6.15: Region of interest: % of guarded nodes in the circuit.

6.1.4 Case Study II: Using the Verifier To Find real fault

In this section, we demonstrate the use of the grind verifier to identify an HLS bug reported in commit $\#faefda$ of IR [54].

Cases study setup: For HLS framework we picked IR . IR [54] is an open-source HLS tool that supports high-level languages such as C/C++. It allows users to optimize the accelerator at the structural IR and generates the hardware accelerator as a dynamic dataflow using the nodes described in Section chapter 2. In the backend, IR uses an open-source library of hardware components, and the designer can extend and incorporate the new modules in the main design as long as the nodes use dynamic interface to talk with other nodes.

We realized that this IR commit had a non-deterministic cache bug that was only triggered under specific conditions. Here we describe how the bug was tracked down. During a run with Machsuite [50], we found that the FFT benchmark failed and produced incorrect values for certain memory locations non-deterministically. The challenge was that there were multiple memory operations any of which could be faulty. We used another application, Stencil2D, from Machsuite [50] to

	State-of-the-art [56, 10]	grind
Target	C only	H-RTL, HLS and C
Monitoring	Offline (post-execution)	Online (in execution)
Scope	User [10], Coarse [10, 56], Fine [56]	Whole accelerator (including blackbox RTL)
Region-of-interest	Wide. Buggy and Correct RTL statements	Focused. Only RTL statements dependent on bug.
Patching	—	Replaces buggy values with golden.
Accuracy	User	Iterative analysis.
Hard failues	Yes. Restricts trace information	No. grind patches values.
Multiple bugs	No. Errors impact other ops.	. Guards patch to isolate error during execution.
DUT Size	Small	Large

Table 6.4: State-of-the-art vs grind for verifying H-RTL

check if the bug is benchmark agnostic i.e., Load component faulty. Interestingly, this revealed that the deviant memory behavior was triggered only in FFT. To further narrow it, we activated guards on all Load components. Since the grind verifier supports patching, in a single run we identified which one of these loads was erring. In this case, only one load was faulted. Since others functioned correctly, we isolated the fault to be external to the load itself. Following this, we instrumented and guarded only the flagged load, and turned on multiple guards that analyzed all the incoming and outgoing ports between the erring load and the cache. This detected that the return cache value was incorrect; however only on specific cycles. Upon further investigation of guards, it turned out that cache had a register overwritten when you had a read miss followed by a write. This highlights the benefits of the grind verifier over asserts; we can let the patched execution continue to help us identify the temporal pattern behind a particular bug. Asserts would simply kill the execution at the load, leaving us with no further information.

Design Setup: In Figure 6.16 shows the overall system design that we used to run the experiment. The system consists of an accelerator core, which is generated by HLS, which is the design that we want to verify. The accelerator is interfacing with a cache that is connected to the main memory using the AXI interface. To enable verifying, grind injects the guard nodes inside the accelerator core alongside the metadata streamed in. The Verifier shares the AXI bus with cache through an arbiter. Furthermore, there is a control unit inside the design that allows the host core to control the accelerator. The verifier writer is connected to the accelerator core using Bore connections as a hardware compiler pass in FIRRTL. We also provide a memory allocator that allocates memory for dumping verified traces in the main memory.

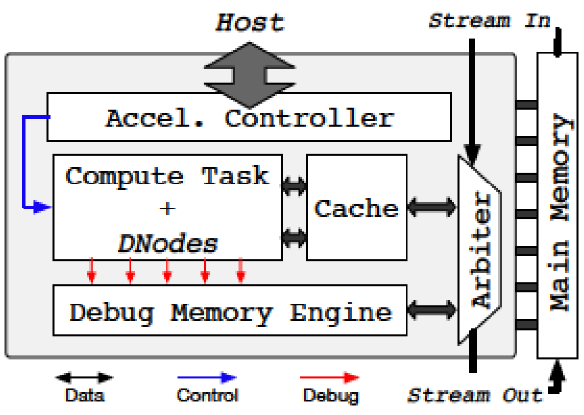


Figure 6.16: Design Setup

6.2 H-RTL profiler and H-RTL faulty evaluation

In this section we look into the advantages of the two additional tools H-RTL profiler and H-RTL faulty. We first compare the advantages and memory uasge of a dynamic profiler with state-of-the-art. And then use H-RTL faulty to inject faults into our benchmarks and observe the effects.

6.2.1 H-RTL profiler vs state-of-the-art

We quantitatively compare our dynamic profiling against prior state-of-the-art (trace-based). We compare the following profilers i) Baseline: trace values to DRAM and post-process in software ii) ACT: profile values when another Boolean signal indicates the operation is active (this is representative of state-of-the-art). iii) HW: Profile pipeline signals iv) MEM: Profile the memory addresses when the operations are active. v) ADDER: Profile compute operations. Figure 6.17 shows DRAM traffic reducing in μgrind vs state-of-the-art [33] and Figure 6.18 illustrates μgrind SRAM usage vs state-of-the-art [33]. μgrind shows a promising reduction. We observe that:
Guard-based profiling saves 200x—35000x times DRAM traffic by building in the profiler on-chip at the site of the H-RTL signal and eliminating the need to write to DRAM.

Guard-based profiling saves on-chip SRAM by helping the user rapidly create dynamic profilers that auto-instrument only the regions of interest.

6.2.2 H-RTL faulty evaluation

Our methodology for evaluating fault injection consists of three steps: 1)*Where to inject bugs?* Using the HLS compiler, we randomly picked 10% nodes of each application of any one of these classes. computation, control, or memory. 2)*What is the bug?* We select each node's output (the node can have multiple outputs) and the error value to inject that output. 3)*Error injections runs:* We inject one fault in the H-RTL circuit, and monitor for crashes, deadlock, and output corruption.

In Figure 6.19 we plot the distribution of faults for each type of error. *FFT* is a memory-intensive application and as a result, the probability of showing a memory bug compared to other applications

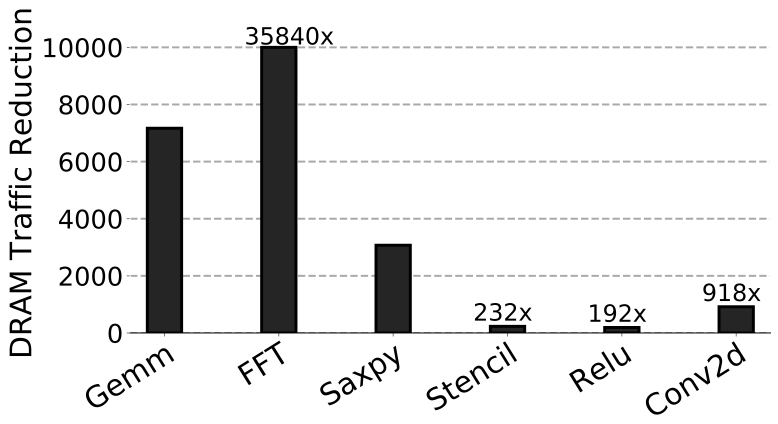


Figure 6.17: DRAM traffic.   grind vs state-of-the-art [33]

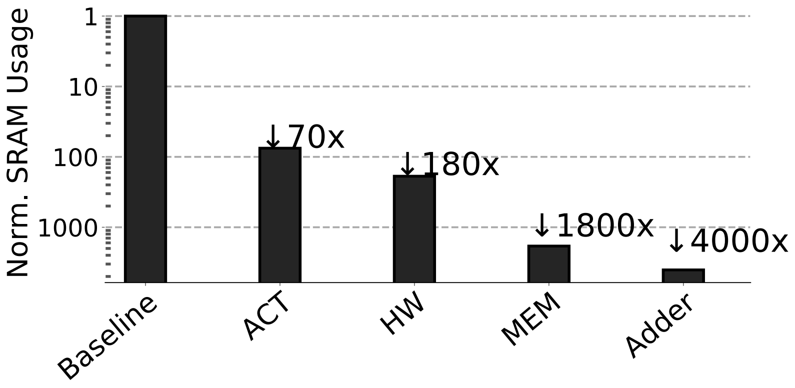


Figure 6.18: Normalized SRAM.   grind vs state-of-the-art [33]. Lower bar means more reduction in SRAM

is higher. In *Conv* and *Gemm*, since the control flow in the application is more complex compared to the other applications, it is more likely that a new bug in the circuit introduces a control type of bugs. This information can give insight to HLS compiler designers as to where to start debugging a faulty change in the compiler.

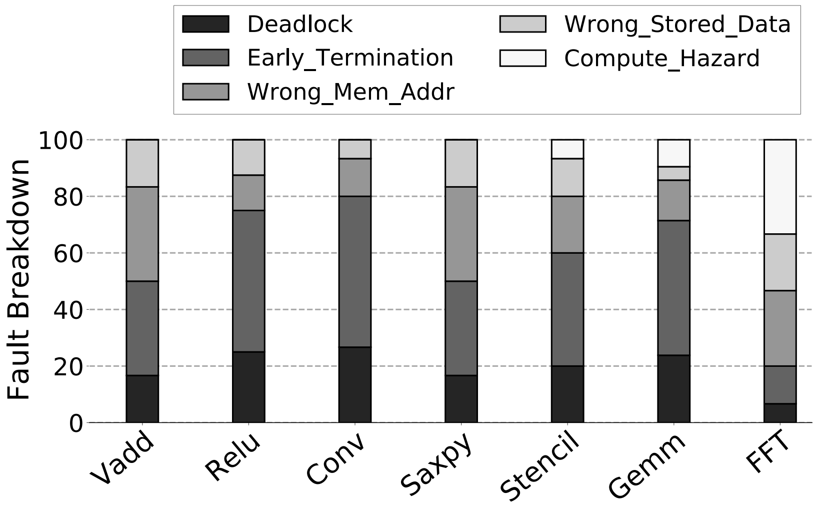


Figure 6.19: Outcome of Fault injection

Bibliography

[1] Boringutils. `https://github.com/freechipsproject/chisel3/blob/master/src/main/scala/chisel3/util/experimental/BoringUtils.scala`. (Accessed on 04/17/2020).

[2] Vivado Design Suite. `https://www.xilinx.com/products/design-tools/vivado.html`.

[3] Vivado hls co-simulation. xilinx.com/t5/High-Level-Synthesis-HLS/Vivado-HLS-Co-simulation-Waveform-signals-never-change/m-p/984041.

[4] High level synthesis with a dataflow architectural template, 2016.

[5] Shlomi Alkalay, Tamas Juhasz, Puneet Kaur, Sitaram Lanka, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Andrew Putnam, Raja Seera, Rimon Tadros, Hari Angepat, Jason Thong, Lisa Woods, Derek Chiou, Doug Burger, Adrian Caulfield, Eric Chung, Oren Firestein, Michael Haselman, Stephen Heil, Kyle Holohan, and Matt Humphrey. Agile Co-Design for a Reconfigurable Datacenter. In *the 2016 ACM/SIGDA International Symposium*, pages 15–15, New York, New York, USA, 2016. ACM Press.

[6] Gilbert Bernstein, Ross Daly, Jonathan, Ragan-Kelley, and Pat Hanrahan. What are the semantics of hardware? In *Workshop on Languages Tools and Techniques for Accelerator Design*, 2021.

[7] Thomas Bollaert. Catapult High-Level Synthesis. In *High-Level Synthesis*, pages 29–52. Springer, 2008. `https://www.mentor.com/hls-lp/catapult-high-level-synthesis/`.

[8] Pavan Kumar Bussa, Jeffrey Goeders, and Steven JE Wilton. Accelerating in-system fpga debug of high-level synthesis circuits using incremental compilation techniques. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[9] Nazanin Calagar, Stephen D Brown, and Jason H Anderson. Source-level debugging for fpga high-level synthesis. In *2014 24th international conference on field programmable logic and applications (FPL)*, pages 1–8. IEEE, 2014.

[10] Nazanin Calagar, Stephen D. Brown, and Jason H. Anderson. Source-level debugging for FPGA high-level synthesis. pages 1–8. Technical University of Munich (TUM), 2014.

[11] Kevin Camera and Robert W. Brodersen. An integrated debugging environment for FPGA computing platforms. In *Proc. of the FPL*, pages 311–316, 2008.

[12] Keith Campbell, Leon He, Liwei Yang, Swathi Gurumani, Kyle Rupnow, and Deming Chen. Debugging and verifying soc designs through effective cross-layer hardware-software co-simulation. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[13] Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, Jia Jun Qin, Mark Aldham, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. From software to accelerators with LegUp high-level synthesis. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–9. IEEE, 2013.

[14] Tao Chen, Shreesha Srinath, Christopher Batten, and G Edward Suh. An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 55–67. IEEE, 2018.

[15] Jongsok Choi, Stephen Dean Brown, and Jason Helge Anderson. From pthreads to multicore hardware systems in legup high-level synthesis for fpgas. *IEEE Trans. VLSI Syst.*, 25(10), 2017.

[16] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross G Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. Type-directed scheduling of streaming accelerators. In *Proc. of the 41st PLDI*, pages 408–422, 2020.

[17] Pietro Fezzardi, Michele Castellana, and Fabrizio Ferrandi. Trace-based automated logical debugging for high-level synthesis generated circuits. In *Proc. of the 33rd ICCD*, 2015.

[18] Pietro Fezzardi and Fabrizio Ferrandi. Automated bug detection for pointers and memory accesses in High-Level Synthesis compilers. In *Proc. of FPL*, 2016.

[19] Pietro Fezzardi, Marco Lattuada, and Fabrizio Ferrandi. Using efficient path profiling to optimize memory consumption of on-chip debugging for high-level synthesis. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–19, 2017.

[20] Jeffrey Goeders and Steve J.E. Wilton. Using dynamic signal-tracing to debug compiler-optimized HLS circuits on FPGAS. In *Proc. of the FCCM*, pages 127–134, 2015.

[21] Jeffrey Goeders and Steven J E Wilton. Quantifying observability for in-system debug of high-level synthesis circuits. In *Proc. of the FPL*, 2016.

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

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

[24] Jeffrey B Goeders and Steven J E Wilton. Effective FPGA debug for high-level synthesis generated circuits. In *FPL*, 2014.

[25] K Scott Hemmert, Justin L Tripp, Brad L Hutchings, and Preston A Jackson. Source level debugger for the sea cucumber synthesizing compiler. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, pages 228–237. IEEE, 2003.

[26] Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson. An empirical study of the reliability of high-level synthesis tools. In *Proc. of the FCCM (Short Paper)*, 2021.

[27] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 209–216. IEEE, 2017.

[28] Al-Shahna Jamal, Eli Cahill, Jeffrey Goeders, and Steven JE Wilton. Fast turnaround hls debugging using dependency analysis and debug overlays. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(1):1–26, 2020.

[29] Al Shahna Jamal, Jeffrey Goeders, and Steven J.E. Wilton. An FPGA overlay architecture supporting rapid implementation of functional changes during on-chip debug. In *Proc. of the FPL*, 2018.

[30] Weng Jian, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. DSAGEN: Synthesizing Programmable Spatial Accelerators. In *Proc. of the 47th ISCA*, pages 268–281, 2020.

[31] Gangwon Jo, Heehoon Kim, Jeesoo Lee, and Jaejin Lee. SOFF: An OpenCL High-Level Synthesis Framework for FPGAs. In *Proc. of the 47th ISCA*, pages 295–308, 2020.

[32] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *Proc. of the FPGA*, 2018.

[33] Sagar Karandikar, Albert Ou, Alon Amid, Howard Mao, Randy Katz, Borivoje Nikolić, and Krste Asanović. Fireperf: Fpga-accelerated full-system hardware/software performance profiling and co-design. In *Proc. of the ASPLOS*, 2020.

[34] Brucek Khailany, Evgeni Khmer, Rangharajan Venkatesan, Jason Clemons, Joel S Emer, Matthew Fojtik, Alicia Klinefelter, Michael Pellauer, Nathaniel Ross Pinckney, Yakun Sophia Shao, Shreesha Srinath, Christopher Torng, Sam Likun Xi, Yanqing Zhang, and Brian Zimmer. A modular digital VLSI flow for high-productivity SoC design. In *Proc. of DAC*, pages 1–6, New York, New York, USA, 2018. ACM Press.

[35] Donggyu Kim, Christopher Celio, Sagar Karandikar, David Biancolin, Jonathan Bachrach, and Krste Asanović. Dessert: Debugging rtl effectively with state snapshotting for error replays across trillions of cycles. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 76–764. IEEE, 2018.

[36] Youngsik Kim. *Formal Verification of High-Level Synthesis with Global Code Motions*. PhD thesis, 2007.

[37] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *Proceedings of the PLDI*, 2018.

[38] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen. Are we there yet? a study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):898–911, 2019.

[39] Chris Lattner. Many traditional llvm people are confused about what graph/dataflow semantics means, they've spent a bunch of time working with imperative execution domains., 2020.

[40] Andreas Lööw. *Lutsig: A Verified Verilog Compiler for Verified Circuit Development*. 2021.

[41] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. Tapas: Generating parallel accelerators from parallel programs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 245–257. IEEE, 2018.

[42] Cristian Mattarei, Makai Mann, Clark Barrett, Ross G Daly, Dillon Huff, and Pat Hanrahan. Cosa: integrated verification for agile hardware design. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–5. IEEE, 2018.

[43] Joshua S Monson and Brad Hutchings. New approaches for in-system debug of behaviorally-synthesized fpga circuits. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6. IEEE, 2014.

[44] Joshua S Monson and Brad L Hutchings. Using source-level transformations to improve high-level synthesis debug and validation on fpgas. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 5–8, 2015.

[45] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li 0002, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types. In *Proc. of the 41st PLDI*, pages 393–407, 2020.

[46] Travis W. Pouarz and Vaibhav Agrawal. 2017.

[47] Raghu Prabhakar, David Koeplinger, Kevin J Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating Configurable Hardware from Parallel Patterns. In *Proc. of the 21st ASPLOS*, 2016.

[48] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Stephen Brown, and Jason Anderson. The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 89–96, April 2013.

[49] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[50] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *Proc. of the IISWC*, Raleigh, North Carolina, October 2014.

[51] Aurélien Ribon, Bertrand Le Gal, Christophe Jégo, and Dominique Dallet. Assertion support in high-level synthesis design flow. In *FDL 2011 Proceedings*, pages 1–8. IEEE, 2011.

[52] Samuel Rogers, Joshua Slycord, Mohammadreza Baharani, and Hamed Tabkhi. gem5-SALAM: A System Architecture for LLVM-based Accelerator Modeling. In *Proc. of the 53rd MICRO*, pages 471–482, 2020.

[53] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 97–108. IEEE, 2014.

[54] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. ir-an intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 940–953, 2019.

[55] Xilinx. `https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/BUG-report-HLS-chooses-the-wrong-II-and-the-result-is-wrong/m-p/1070935`, 2020.

[56] Liwei Yang, Swathi Gurumani, Deming Chen, and Kyle Rupnow. Autoslide: Automatic source-level instrumentation and debugging for hls. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–130. IEEE, 2016.

Appendix A

Code

A.1 Inserting guards

In this section, we look into the actual steps to add a single verifier guard in the circuit. To extract the data and dependence graph from the LLVM in the IR project follow the instructions below:

```
cd muir/build/scripts
source setup.sh
cd /muir/build/tests/c
make clean
make
```

This will create the JSON files. The files with the names benchmarkname.muir.JSON contain the dependence garph information.

The steps to insert a guard in the original circuit are as follows: First we change the platform from IR to IR-sim which is a driver that performs the simulation for IR. In this example we choose the Relu benchmark to instrument, first we open the file:

```
vim /muir-sim/examples/relu_config.JSON
```

This file is a config file for the simulation driver. Let's say we have chosen a node with the ID of 16 to be guarded. The config file has to be updated to show that the number of guards is set to 1 and the list of guard IDs is equal to 16. This information then is passed to the Reader/Writer wrappers to set the connections.

```
"Accel" : {
    "nameAccel" : "relu",
    "numPtrs"   : 2,
    "numGuard"   : 1,
    "numVals"   : 1,
            ...
    "boreIDs"   : [16],
            ...
```

In the generated Scala code, the Guard enable flag has to be set to true and a golden-value file has to be passed to the node that is being guarded:

```
vim muir-sim/hardware/chisel/src/main/scala/generator/relu.scala
```

```
        ...
 val binaryOp_inc1316 = Module(new ComputeNode(NumOuts = 2, ID = 16, opCode = "
    add")(sign = false, Guard = true, GuardVals=GuardReader("relu.dbg"){16}))
        ...
```

Make sure that this file, containing the golden values is present in the directory. The first column is the IDs and the next ones are the values.

```
cd muir-sim/hardware/chisel/src/main/resources/guards/
vim relu.dbg
```

Finally, run and simulate the instrumented circuit with the followiwng command:

```
cd muir-sim
python3.7 run.py --accel-config examples/relu_config.JSON
```

A.2 Verifier implementation

Now that we know how guards can be manually added to the circuit the idea of the iterative verifier being fully automated seems simple. The verifier takes following steps:

- Add a "benchmarkname".dbg file containing all the golden values to the path: *muir   sim hardware chisel src main resources guards*
- Enable guards for the initial list using the node IDs modifying the JSON and Scala files.
- Simulate and run.
- Update the guard list based on the dependence graph JSON file, if a node is faulty, add its parent nodes to the list. if not remove node ID from the list.
- Repeat until there are no more parent nodes in the basic block to inspect.