mu-grind: A Framework for Dynamically Instrumenting HLS-Generated RTL

Parmida Vahdatniya, Amirali Sharifian, Reza Hojabr, Arrvindh Shriraman
School of Computing Sciences, Simon Fraser University
https://github.com/sfu-arch/muir-lib
Canada
{parmida_vahdatniya,amiralis,shojabro,ashriram}@cs.sfu.ca

Abstract

High-level synthesis compilers (HLS) enable the rapid creation of accelerator circuits. Unfortunately, compiler generated RTL (H-RTL) is inconsistent in terms of quality, hard to comprehend, and tends to be brittle [28, 41]. This paper develops a framework to help HLS compiler architects inspect and profile H-RTL. Prior state-of-the-art tools [23, 57] have predominantly focused on tracing. Tracing requires massive amount of on-chip buffering, limits the H-RTL design size, and only support post-mortem analysis at the end of the execution.

We propose *µgrind*[1], a dynamic instrumentation framework for H-RTL. The key technique is guards, additional logic that we auto-inject into the output of HLS compilers (H-RTL). Guards perform two tasks: i) they run analysis functions on the values fed from the H-RTL signal, and ii) patch values into the H-RTL during live execution. Guards can either be mapped onto the FPGA or can be co-simulated along with the H-RTL. *µgrind* can remove them once the H-RTL is finalized. Leveraging *µgrind*, we create a novel tool, *H-RTL checker*, that precisely identifies the erring signal and cycle without any user involvement. Compared to prior art, *µgrind* requires 2—10× less SRAM, supports 5× larger H-RTL circuits (upto 98% of the FPGA) and completes checks in <24 hours (including FPGA synthesis time). We also develop two additional tools: i) *H-RTL faulty*, which deploys heterogeneous guards to study circuit resilience, and ii) *H-RTL profiler*, which creates detailed execution histograms. We save between 200-35000X DRAM traffic compared to prior art, by avoiding traces.

CCS Concepts

• **Computer systems organization** → **Architectures**; • **Hardware** → **Hardware-software codesign**.

Keywords

Debuggers, High-level synthesis, FPGAs, Hardware instrumentation, Dynamic instrumentation

ACM Reference Format:
Parmida Vahdatniya, Amirali Sharifian, Reza Hojabr, Arrvindh Shriraman. 2022. mu-grind: A Framework for Dynamically Instrumenting HLS-Generated RTL. In ,. ACM, New York, NY, USA, 13 pages. https://doi.org/https://doi.org/10.1145/3559009.3569671

[1]The name is inspired by binary instrumentation tool Valgrind.

PACT '22, October 10–12, 2022, Chicago, IL, USA

ACM ISBN 978-1-4503-9868-8/22/10…$15.00
https://doi.org/https://doi.org/10.1145/3559009.3569671

1 Introduction

High-level synthesis research and development is error prone. Software design is comparatively straightforward....with mature debugging tools. — **Andrew Canis, CTO, Legup HLS** [15, Page 8]

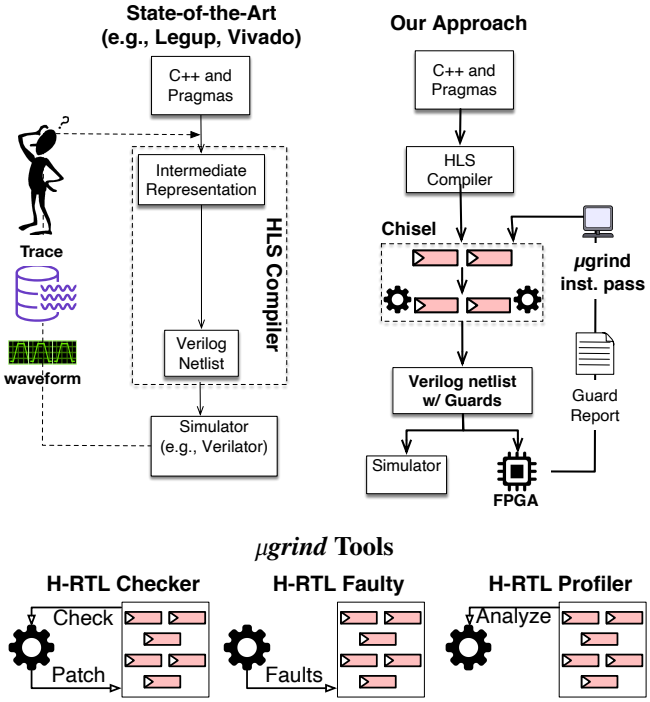


Figure 1: Overview of *µgrind* toolflow.

The last few years has seen a surge in research into high-level-synthesis compilers (HLS) that auto-translate high-level languages into RTL [5, 19, 33, 34, 43, 46, 48, 51–53]. Today, computer architects actively develop HLS compilers and deploy it for creating custom IP targetting both FPGAs [1] and ASICs [38]. It is widely acknowledged that the key impediment in HLS is the opaqueness of the compiled or generated RTL (H-RTL) [41]; even industry-standard HLS break the H-RTL in eccentric ways [28].

A leading HLS expert cites the lack of mature tools to inspect HLS generated output (H-RTL) as being a key hindrance [15, Page 8] [41]. Multiple tools exist (e.g., valgrind, Dynamorio, gcc -p) to analyze the output of software compilers (i.e., binaries). However, HLS compilers lack such a framework. The question we answer in this paper: **how can we help the HLS compiler developers instrument the generated RTL with low-effort and in a flexible manner to understand the dynamic execution of hardware (see Figure 1).** Our goals are

three-fold i) low-effort instrumentation 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. ii) flexible instrumentation i.e., we need a configurable framework that adds in additional logic and SRAM only for the signals instrumented in the H-RTL. Conversely, it should be easy to remove the instrumentation entirely from the H-RTL, once the accelerator is analyzed. iii) dynamic instrumentation i.e., the instrumentation should be able to analyze and modify signals during the execution. We demonstrate that live execution analysis is essential to creating checker tools that avoid muddled-up logs.

Waveforms are the most prevalent approach [3]. This requires the user to inspect a verbose H-RTL netlist across potentially millions of simulation cycles [39]. State-of-the-art tools help annotate waveforms with additional information [12, 24, 27, 31, 57]. However, this leaves open the question of how can a user know which portion of the opaque H-RTL to focus on. Some works have provided a gdb-like environment [11, 13, 14, 25, 32] to inject kill switches. The kill switches are similar to asserts and stop the circuit [50]). They do not support analysis of the live execution.

Our Approach

We propose *µgrind*, a framework for dynamically instrumenting HLS generated RTL (Figure 1). The key technique is *guard*s, circuitry that we mix into the H-RTL to tail any register, memory entry, and signal. *µgrind* builds on modern RTL toolchains (Chisel [30]) to add and remove *guard*s. Guards get mapped onto the FPGA prototype with H-RTL; they can also be co-simulated in verilator. During execution, guards can dynamically extract, run analysis logic, and modify (or patch) the H-RTL's signals. This eliminates the need to trace a verbose dump of signals to the DRAM for post-execution analysis. Guards only write post-analysis data to the DRAM. This saves DRAM traffic, reduces on-chip SRAM, and enables larger circuits to be analyze. Finally, patching prevents the unfettered propagation of erroneous signals during debug runs (unlike asserts), enabling us to converge rapidly on region of analysis. Guards run concurrently which reduces the overhead typically associated with instrumentation. The dynamic term refers to the fact that: i) we can analyze internal hardware signals during runtime (a first), and ii) we can turn off the instrumentation at runtime.

We create three tools to demonstrate *µgrind*. a) *H-RTL Checker:* a novel checker, that pinpoints the statements and cycles in which the H-RTL deviated from the software behavior. We exploit the key observation that during the checking stage the majority of the H-RTL circuit typically functions correctly. It uses a smaller H-RTL circuit to rapidly eliminate false guards. Once we have narrowed down the guard's region of interest, we scale up the design-under-test to 90% of the FPGA. This way we can check complete real accelerators in <24hrs. We also study two other tools: a) *H-RTL Faulty :* leverages *guard*s to check circuit resiliency. It injects a variety of faults (e.g,. struck control) into specific H-RTL signals, while simultaneously monitoring the circuit. c) *H-RTL Profiler :* a tool that builds into hardware the logic for histograming and summarizing H-RTL activities.

- We are the first to propose techniques for hardware instrumentation. *µgrind* enables a tool to extract H-RTL signals and attach user-defined analysis functions. FPGA prototyping demonstrates that instrumentation imposes limited overhead, 10—15% extra logic and ≃ 5% Mhz penalty.
- We develop a novel tool to find bugs in H-RTL and automatically refine regions of inspection. Compared to state-of-the-art, we find bugs in RTL 5× larger, in under 24hrs (including FPGA synthesis).
- We demonstrate that iterative approach to debugging is practical. For accelerator circuits, we can narrow down bug site within a few iterations (3-4 iterations 16-24hr) and we can study designs up to 90% of FPGA (prior state-of-the-art restricted to 20% of FPGA).
- Compared to prior trace tools, we demonstrate that dynamic instrumentation can save 200—35000× DRAM traffic and 2—10× of on-chip SRAM.

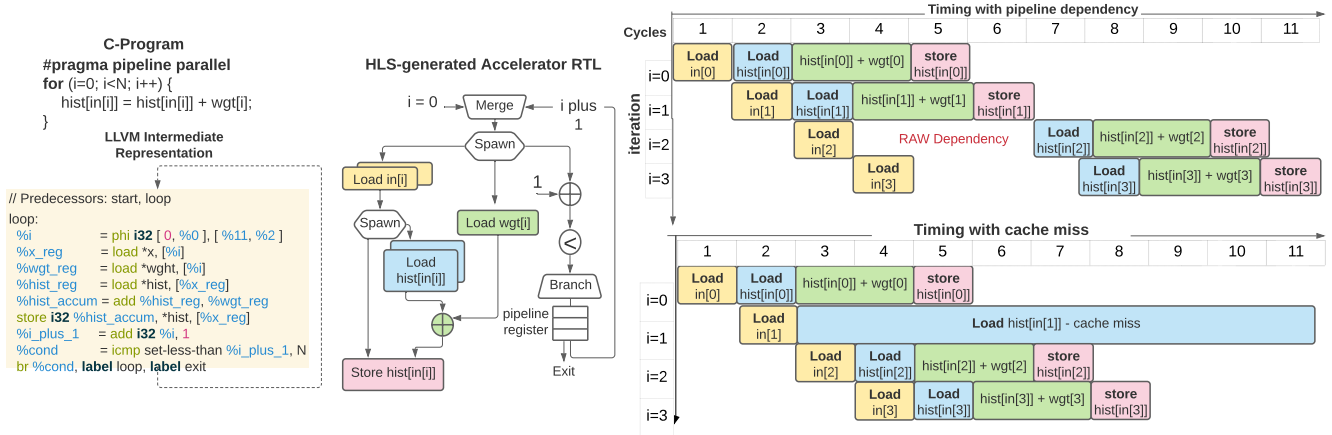


Figure 2: Overview of HLS translating histogram kernel [35, 53]. Left: HLS inputs. C program and SSA IR. Mid: HLS Output and Accelerator H-RTL. Right: Dynamic schedules and execution pipeline.

2 Motivation and Scope

First, we provide an overview of an HLS compiler and H-RTL (Figure 2). State-of-the-art HLS compilers [35, 53] translate C/C++ to a token-based dataflow circuit. Here we show a histogam kernel. The execution is dynamically-scheduled i.e., the nodes and operations are triggered as dependencies are satisfied, and no centralized FSM (finite state machine) is required. The pipeline diagram illustrates the timing. Within the loop, a data dependency may exist between the load of `hist[in[i+1]]` and the store to `hist[in[i]]` of a prior iteration, depending on the contents of `in[]`. When the dependency does not occur, the dynamic schedule initiates a new iteration each cycle. When a dependency does occur, the dynamic schedule stalls the pipeline to satisfy read-after-write dependency. Loop iterations can complete in arbitrary order. When the underlying loads and stores are connected to a cache it leads to non-determinism in the latency and the timing of memory operations. For instance the add operation in iteration 2 is stalled until the load of `hist[2]` from cache completes. In the meanwhile, the third iteration starts and completes.

2.1 H-RTL instrumentation vs. Binary instrumentation

HLS compiler developers cite the lack of fixed state as to why H-RTL requires new techniques to binaries [10, 42]. **i) Executable (binary) vs. Structural (H-RTL):** A binary runs on existing hardware. The instrumentation reads and writes ISA-visible registers/memory and runs on same hardware as the binary. The state is accessed via the processor instructions. H-RTL describes a microarchitecture structure and accelerators are based on dataflow. We have to create functions units, bind operations, and physically route values. Only recently RTL toolchains have made it possible to programmatically edit H-RTL [30]. **iii) Centralized fixed state (binary) vs Distributed, variable state (H-RTL)** Finally, any instrumentation framework needs to read and write state from the target. With binary, the ISA registers and memory state are defined and centralized i.e. all binaries refer to common ISA register state. HLS compilers customize the state for each H-RTL accelerator and distributes state across in the pipeline latches, operand buffers, and scratchpads. iii) **ii) Imperative ISA (binary) vs Concurrent Dataflow (H-RTL)** A binary is an imperative specification in a target ISA. The instrumented binary implicitly supports sequential semantics enforced by the underlying cpu. H-RTL is a concurrent specification in which ordering of operations has to be defined by *µgrind*.

2.2 Motivational tool: Checking H-RTL errors.

To motivate how instrumentation can help with HLS compiler research, we briefly preview a checker tool from Section 5). We track the git commits in a state-of-the-art HLS compiler [53](*µIR*) and find errors introduced due to H-RTL passes. We discuss the errors and motivate the need for instrumentation that can track signal values and cycle timing. We communicated with the authors of *µIR* and verified the cause of these errors [18, 43, 53].

H-RTL Error 1: Stuck Control

Detection: Instrument the merge mux's output signal.

Many HLS compilers translate LLVM's SSA representation to RTL (e.g., LLVM IR [2, 35, 53]). LLVM periodically updates the SSA syntax during major releases. In this instance LLVM reversed the order of labels in the select and ϕ ops. This led the HLS to wire the mux data lines to the merge node in the incorrect order (see Figure 3). Due to the mix-up, the mux is stuck at and always propagates i=0 on

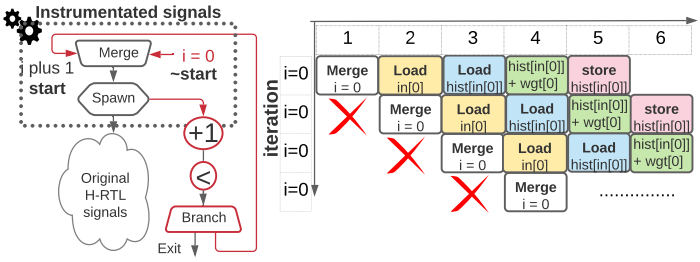


Figure 3: H-RTL Error 1: Stuck control caused by LLVM syntax mismatch leading to incorrect mux wiring.

each iteration of the loop; the loop keeps re-executing iteration i = 0. Tracing or waveforms cannot catch this bug since execution will never terminate. *µgrind*'s dynamic instrumentation will capture the output of the merge and analyze if the loop induction variable.

H-RTL Error 2: Incorrect dataflow pipelining

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

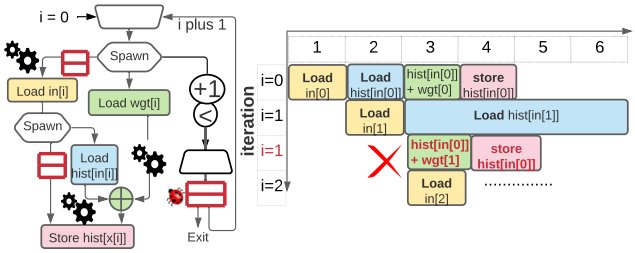


Figure 4: H-RTL Error 2: Incorrect pipeline buffer depth setting leading to faulty operands.

These classes of errors are reported even by Xilinx's Vivado [56]. 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 leading to `hist[in[0]]+wgt[1]`; 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 iteration index, and the adder operands.

HLS Error 3: Faulty Cache Handshaking

Detection: Instrument the cache request and response lines, and check number of requests/responses.

A common cause of error is the interface to the shared cache (or scratchpad). Typically the cache or scratchpad is a blackbox IP invisible to HLS. The HLS statically schedules loads and store across latency-sensitive request and response ports. In this particular case, *µIR* HLS incorrectly scheduled the `load hist[in[i]]` on the same cycle as another load. This led to a load being missed missed by the cache. *µIR* HLS [53] also reported similar error causing incorrect response errors due to wrong address. *µgrind* 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 type of checkers can be since *µgrind* permits the user to define analysis function within the guards.

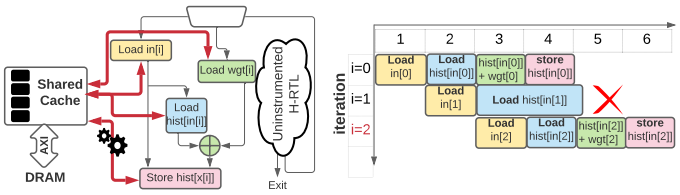


Figure 5: H-RTL Error 3: Incorrect interfacing between H-RTL and Cache leading to missed request and circuit lockup.

2.3 Complexity of instrumented H-RTL circuits.

We now motivate the need for a framework that can automatically inject instrumentation into H-RTL (without requiring any human editing). We study end-to-end applications from Machsuite [49] *Relu, Saxpy, Vadd, Conv2D, Stencil, and Gemm*. Upto 40K lines of verilog, 32 state FSMs, 400 modules, 40 stage pipeline, and 700 operations in a single cycle. Table 1 lists the characteristics of the H-RTL circuits. For the interested reader, the H-RTL imported into Chisel can be viewed here (https://anonymous.4open.science/r/d6f70aaf-3014-4353-9b48-cc5759080898/).

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 1: RTL Complexity of *guard*ed Accelerators

We use four proxy metrics i) **Verilog LOC:** The number of lines of verilog strongly correlate with the number of H-RTL variables (signals or registers). ii) **Ctrl-states** This measures the complexity of the control FSM. Accelerator with nested loops, require multiple states. iii) **Verilog modules:** The number of modules instantiated; each module roughly corresponds to a dataflow operation. The higher the number of modules, the more the motivation for instrumentation since waveforms tend to be polluted. iv) **Pipeline depth:** In HLS, the pipeline can be much deeper than CPUs i.e., more instruction execution overlap. This makes it hard to analyze timing-dependent errors. v) **Concurrency:** The H-RTL circuits we investigate are highly concurrent with upto 700 parallel fine-grain ops; an instrumentation framework is required to narrow the region.

3 *µgrind*: Architecture and Design

3.1 Auto-Wiring *guard*s into H-RTL

In *µgrind*, the end-user or HLS developer does not need to read or edit the H-RTL. Figure 6 illustrates the passes we have developed to mixin guards into the H-RTL. The example illustrates a simple address checker that analyzes the loads in the H-RTL circuit. In ❶ *µgrind* iterates over the SSA representation within the HLS compiler and creates a mapping table between SSA registers and the verilog modules. This serves two purposes: i) a tool (or user) can indicate their region of interest at the program-level and we can track down the signals to be guarded. ii) we can reverse-map the guard output to the higher-level region of interest using the SSA as an intermediary. In ❷ the guard list is filled based on the instrumentation goals e.g., load nodes. Each entry also includes the verilog module and the analysis function. Each entry can independently determine the guard class i.e., multiple guard classes can be simultaneously active. In ❸ *µgrind* iterates over the H-RTL and identifies the signals (registers and wires) within the module. For each signal, *µgrind* includes an `AddGuard()` annotation in the H-RTL module. In this example, since loads are instrumented, the address and data fields are annotated. ❹ In this stage, we define the guard circuits and connect it to the actual signals. *µgrind* leverages FIRRTL, a compiler that loads H-RTL into a data-structure that we can transform and rewrite. The main challenge is that 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 *µgrind* uses a FIRRTL pass that "bore" through the module hierarchy (Figure 7 illustrates). We add wiring for toggle enabling/disabling the guards from a tool. We also wire in a trigger switch that is enabled when the module is active (e.g., load). Finally we bore the H-RTL signals to the guard and corresponding patch values in the reverse direction.

Guard Internals

Each guard monitors an H-RTL signal and includes five components: i) **Trigger:** a boolean signal that activates the *guard* to pay attention and start analyzing the H-RTL signal. This serves to avoid the data deluge of waveforms. ii) **Shadow RAM** a scratchpad for holding guard's metadata. The metadata is streamed from DRAM during the execution. There could be multiple metadata buffered, and a *guard*

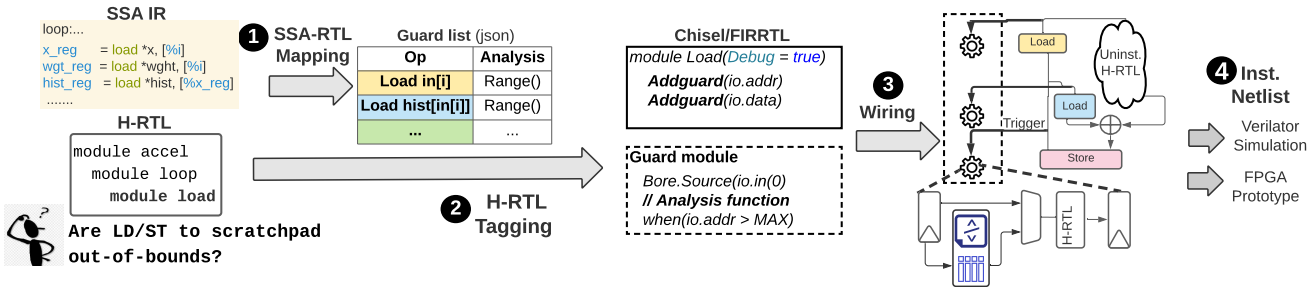


Figure 6: *µgrind* Toolflow

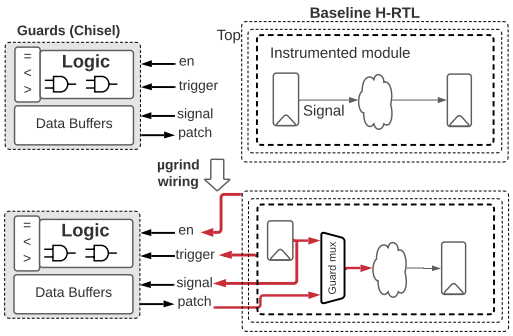


Figure 7: Boring wires between guards and nested H-RTL

may refer to different shadow values depending on execution cycle. iii) **Analysis():** a logic block that processes the incoming H-RTL signals and shadow value. It further records the data and/or modifies the H-RTL on-chip signal (patching). The majority of analysis functions require simple logic, e.g., `isEqual()` or `isRange()` that can be accomplished in 1 cycle. Guards also support multi-cycle analysis functions, since they interface with H-RTL using latency-insensitive connections iv) **Patch value:** The patch value overwrites the H-RTL signal during execution. Patches are useful for fixing erring signals during debugging and injecting faults for testing resiliency. v) **Tracer RAM:** 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 analysis output.

Guard core

A guard core serve as the top module for all the guards mixed in with the H-RTL (Figure 8). Having a separate guard core enables the following benefits: i) guards can share buffers to interface with the DRAM, ii) we 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. iii) guards can exchange information with each other for dynamic analysis. The core collects the results of the analysis and burst them to the main memory in double-buffered batches. 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 *guard*s 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.

4 *µgrind* APIs

Event APIs

The event APIs are included at the top-level of the H-RTL circuit and are used to dynamically turn on (or off) callbacks to guards. The event APIs are triggered by *µgrind* when the module in the H-RTL is activated. These events are module start or termination, and entry/exit of module function units. The H-RTL is represented as a latency-agnostic structural graph. Nodes in the graph represent the compute, control and memory modules. The link here includes all the H-RTL benchmarks evaluated with the relevant event APIs [54]. Code 1 shows excerpts from the top-level of a Relu circuit. Every node or module that has debugging set to true, triggers the guard when the node kick-starts in the dataflow at runtime. We extract the

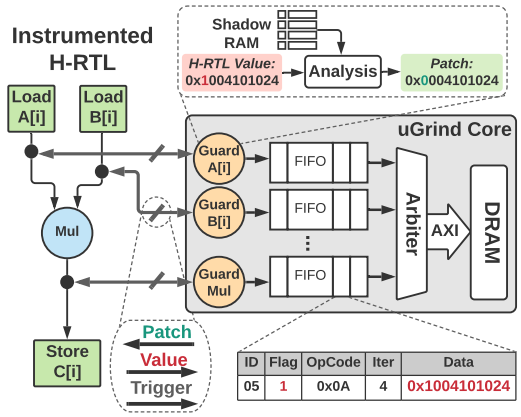


Figure 8: Guard Core

dependencies in the dataflow graph and expose a json file, in which the tool (or user) can mark events of interest(code 2).

```
1 // %mul = mul nsw i32 %shr, %W,
2  val mul3 = Module(new ComputeNode
3  (NumOuts, ID = 3, opCode = "mul", sign = false)
4  (Debug = True))
```

Code 1: Line 101 from Relu H-RTL.

```
1 { "id" : 3,
2  "name" : "mul3",
3  "operands" : ["INS_13", "INS_21"],
4  "bb" : 19,
5  "type" : "Binary" }
```

Code 2: Event configuration (Relu.json)

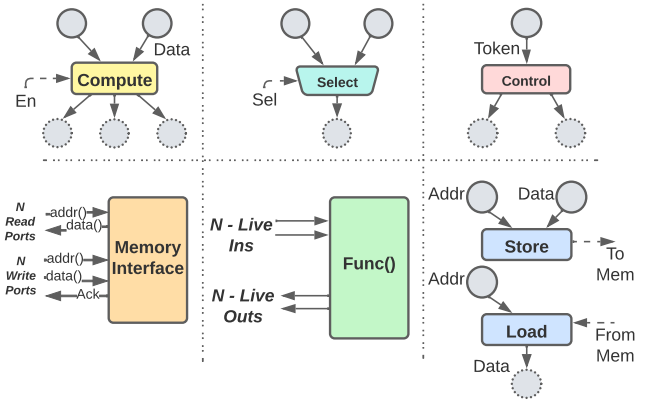


Figure 9: List of nodes with guard callback support

4.1 Signal APIs

Signal API is akin to a hardware callback. During HLS they tunnel wires from signals in the H-RTL to the guard modules. During runtime they expose the H-RTL state to guards. The signal API is hierarchical. The module-level API treats each node in the H-RTL as a blackbox and only extracts the io ports of each module (e.g., the operands and output). The wire API can extract the signals within each node, any register, signal, and RAM entry. The signals extracted by wire API depend on the type of the module (see Table 2). HLS blackboxes (e.g., FPGA specific FPUs or caches) default to the module API. Code 3 illustrates the wiring we automatically add to module for the internal signal. A `sink` (line 3) is the hardware callback from H-RTL module

to guard. The `source` (line 6) wires in guard output back into the H-RTL module. Every source or sink call takes a string name for the wire, which helps identify the corresponding guard partner during hardware synthesis.

Table 2: *µgrind* Signal API for H-RTL nodes.

Components	H-RTL Signals
Compute (INT,FP)	ID, Predicate, Operands, Output
Addr (e.g.,Gep,Ptr)	Base Addr., Offsets, Type
Control (e.g.,Select)	Mux, Enable, Predicates, Branch
Load/Store	Addr., Data, Size, Type (Local,DRAM)
Cache/Scratch	Req./Resp packet, Data, Valid, Data, Address
Function	Arg., Output, Operation ports, Pipeline reg.

```
1  class ComputeNode() {
2  //Callback module to guard
3    addSink(io.FU.data, s"FU_data${ID}")
4    addSink(io.FU.valid, s"FU_valid${ID}")
5  // Analysis result from guard to module
6    addSource(io.Callin, s"Guard_Callin${ID}")
7    addSource(io.Patch, s"Patch_data${ID}")
8  }
```

Code 3: Signal API for boring wires.

4.2 Instrumentation API

µgrind includes a library of guards that the tool or user can declare without any additional effort (Table 3). The user can also create customized guards in Scala and Chisel [8]. Code 4 shows a simple equivalence checker. The APIs are meant to have a software-feel, and *µgrind* will create a hardware circuit and attach it to the H-RTL module. The guard arguments and return values are wired in using the mirror image sink/source calls (like 6). The API requires the tool to declare three components. i) Trigger (line 11): a valid flag that activates the guard. Here we use the function unit (FU)'s valid signal. ii) Analysis (line 12): This is an acyclic function of the signals extracted from the module. Here the isEqual() analysis function checks if the FU output is equivalent to the golden and returns the boolean result. iii) Patch (line 13): The patch calls back in to the module. In hardware we set the callin flag and guard output lines. The patch is gated based on the result of the analysis function i.e., patch output is not sent if the analysis evaluates to false. (line 4:"when()" in RTL is equivalent to an "if statement"). Code 5 illustrates two other guard types i) a profiler which silently analyzes incoming module data, and immediately calls into the module i.e., patch is not gated. ii) a fault injector that does not perform any analysis, but patches in bogus value into the module's output.

```
1  val isEqual(FU_out, golden) = (FU_out == golden)
2  // Guard top module
3  class Checker() extends Guards {
4  val io = {
5    addSink(io.FU.data, s"FU_data${ID}")
6    addSource(io.Guard.out, s"Guard_data${ID}")
7    .....
8  }
9  when(FU_valid) { // Callback from Module
10   // Analysis
11   val result = isEqual(io.FU.data, golden)
12   when(result) { // Patch function
13         io.Patch := io.golden
14   }
15     io.FU.callin := true.B
```

Table 3: *µgrind* instrumentation library.

Guards	Description
`Check`	Check module output is equal to golden value
`Patch`	Check and update module output with golden value
`Assert`	Check and stop circuit, if output does not match
`Activity`	Count the number of times module is active
`Hist`	Create a histogram of output values of module
`isRange`	Check cache or scratchpad request address range
`isValid`	Check if the input operands to a module are valid
`isTarget`	List the modules activated by a control module
`Fault`	Inject a faulty value into the module's output

```
16 }
```

Code 4: Instrumentation API

```
1  // Profiler creates histogram
2  class Profiler extends Guards() {
3    val bins = Vector(Counter(0.U),256) // Histogram
4    // Calculate index of counter.
5    val idx = FU.io.out.signal(31,24) // Index
6    when (io.FU.valid) { // Trigger
7       bins(io.FU.valid).increment() // Analysis.
8    }
9    io.FU.callin := true.B   // No patch.
10 }
11 // Faulty guard injects faults in the module
12 class Faulty extends Guards() {
13   when(FU_valid) {   // Trigger
14     io.FU.callin := true.B    // Always Patch
15     io.FU.out := LFSR() // Faulty value
16   }
17 }
```

Code 5: Profiler and Faulty Guards

5 Tool 1: H-RTL Checker

To demonstrate the utility of *µgrind* we develop a tool that helps with HLS compiler research. The tool performs checking between the H-RTL signals (hardware behavior) and SSA register state (software behavior) for an execution. The goal of the checker is not formal verification [40], but rapid discovery of functional bugs in H-RTL.

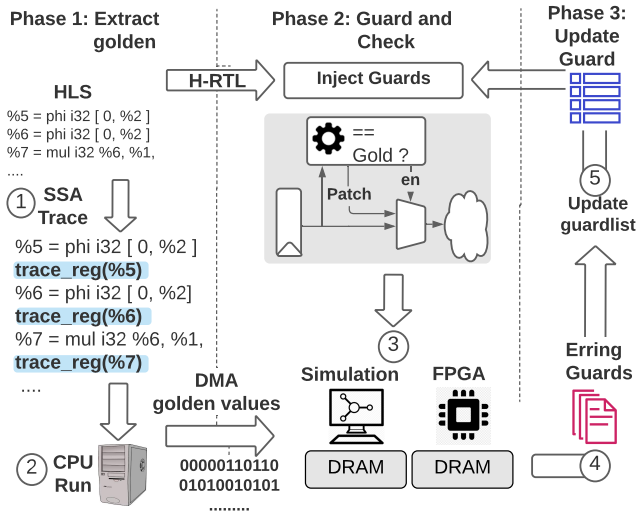


Figure 10: Iterative H-RTL checker tool built using *µgrind*.

Tool Description

Figure 10 illustrates the checker. `Phase 1: Extract golden` is run once. The first requirement in debugging hardware is obtaining a reference golden behavior. ① Here, we modify the HLS compiler [53] to dump the dependence graph and the SSA register list (to a JSON file).

② We insert software instrumentation to trace the SSA register values and memory live-outs from a CPU run. The golden values are written to the DRAM and are consumed by guards during the instrumented FPGA run. ③ The instrumented H-RTL is synthesized and mapped onto the FPGA. In the first iteration, we instrument stores, control, live outs. We will automatically refine the list (in subsequent iterations)

④ The guards write the IDs of the signals that deviate 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 predecessors i.e., we check if the errors originated earlier in the circuit. We keep iterating phases 2 and 3 until we find positively identify the erring signal i.e., the signal whose parents did not err but children did.

Here, we rely on the user to provide a list of inputs based on application knowledge. Test input coverage is not a major concern for H-RTL. The H-RTL is fixed-function and dataflow-based, unlike CPUs that are programmable and include dynamic issue [36]. In H-RTL, inputs have less impact on which portions of the circuit is active/inactive. Also, the dominant overhead is FPGA re-synthesis on each iteration. Hence, generating golden values for multiple inputs is feasible. The primary goal is to minimize the number of iterations, by using backward slicing.

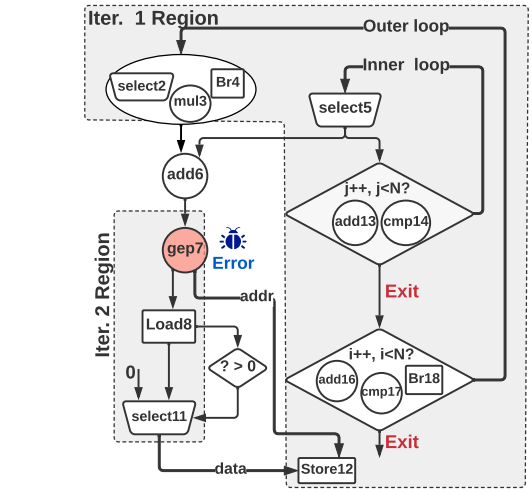
Working Example

Figure 11 shows the the relu kernel, a nested loop. The `Gep7` has a hardware error that causes the faulty value to propagate and taint successors e.g., `Store12`. The goal of the checker is to find the true positive in the midst of multiple false positives i.e., find the erring `gep7` without flagging `store12` and other successors.

In the first iteration, the guard list is initialized to the live-outs and control-nodes in the H-RTL. The guards will check equivalence with golden values and report the `Store12` as erring. In the second iteration, we guard the backward slice of the `Store12`, `Select11` and `Gep7`. In this iteration `Gep7` will fail the checks, but `Store12` will work correctly. The guards on `Gep7` will detect the deviation, note it down, and patch with the available golden value (which was already required to detect the deviation). The patching will prevent the forward slice from becoming tainted and being reported. `Select12` is a false guard as it is functioning correctly and we would not have guarded it, if we had oracle knowledge. The efficacy of the checker is determined by the successive refining and trimming of these false guards.In the final iteration, we guard `Gep7`'s backward slice, `Add6`. Since the `Add6` is not faulty it will pass the checks, and we find the error i.e., `Gep7`.

6 Tool 1 H-RTL Checker Evaluation

In this section, we evaluate the H-RTL checker. We study two forms of deployment: running co-simulation within a Verilog simulator and the other a deployment on Amazon AWS F1, Xilinx UltraScale+



Iter.	Guard list	Patched Guards	Back slice	Fwd. slice	#G
1	mul3, cmp 17, cmp 14, Store12	Store12	select11, gep7	Memory Liveout	4
2	select11, gep7	gep7	add6	Load8, Store12	2
3	add6				1

Figure 11: Illustrating H-RTL checker in a Relu circuit. `gep7` **has an error.**

FPGAs (synthesis results in § 9). All the circuits studied have deterministic bugs occurring at the same site, and triggered at the same cycle. We include a citation to our anonymized benchmark code [54].

(1) **SRAM** (§ 6.1): The resources consumed by the guards determine the resources leftover in the FPGA for the H-RTL. Lower resources, implies we can support larger design.
(2) **Time-to-check**(§ 6.2): This corresponds to the the wall-clock time taken to identify the true-positive (error) or confirm the correctness of the circuit.
(3) **False guard rate**(§ 6.4): The False guard (FG) rate as the fraction of nodes that we checked but eliminated for subsequent checker iterations.
(4) **H-RTL size:** We measure the size of the design as a % of the the FPGA resources occupied. Higher % implies we can check realistic designs that fill the FPGA, without having to scale them down for instrumentation.

§ 6.1 Result 1:*µgrind requires 2—10× less on-chip SRAM than state-of-the-art trace-based checkers [57].*

§ 6.2 Result 2: *µgrind's iterative approach can verify circuits 2—5× larger. We can progressively scale up design size as µgrind's approach narrows site of bug.*

§ 6.2 Result 3: *µgrind demonstrates that iterative checking will be practical. Patching and dynamic checking minimizes number of iterations required to find bug. Despite FPGA resynthesis, bugs uncovered in H-RTL within 16-24hrs (without human involvement).*

§ 6.4 Result 4: *µgrind can rapidly trim the false guards. The FG rate factor for circuits we study is 0.69. i.e., 69% of guards will be eliminated by our backward slice in each iteration. Only 3-4 iterations required to check a circuit.*

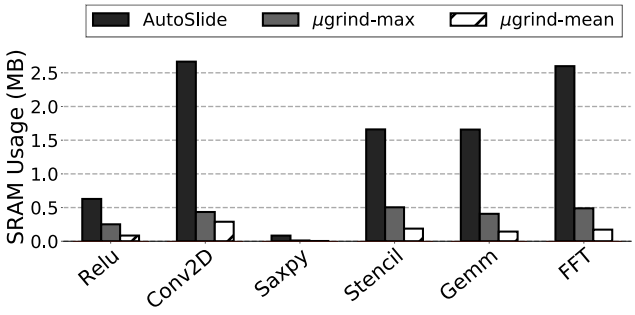


Figure 12: SRAM requirement of *µgrind*

6.1 SRAM: State-of-the-Art vs *µgrind*

Here, we compare the amount of SRAM required against Autoslide [57] the state-of-the art debugger. Overall (Figure 12), *µgrind* has a lower SRAM requirement relative to AutoSlide, while maintaining a low time-to-completion. *µgrind* requires 2–10× less SRAM than trace-based approaches. 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). *µgrind* spreads the checking over multiple iterations. 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 circuit tainted by the unguarded erring signals.

6.2 Time-to-check vs. H-RTL size

In this section, we measure the time-to-check complete accelerators. There is an inherent tradeoff between the time to check and the size of the H-RTL. We support three flows (Figure 13):

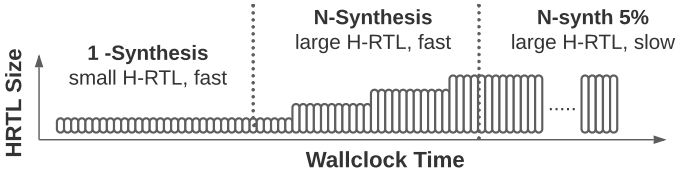
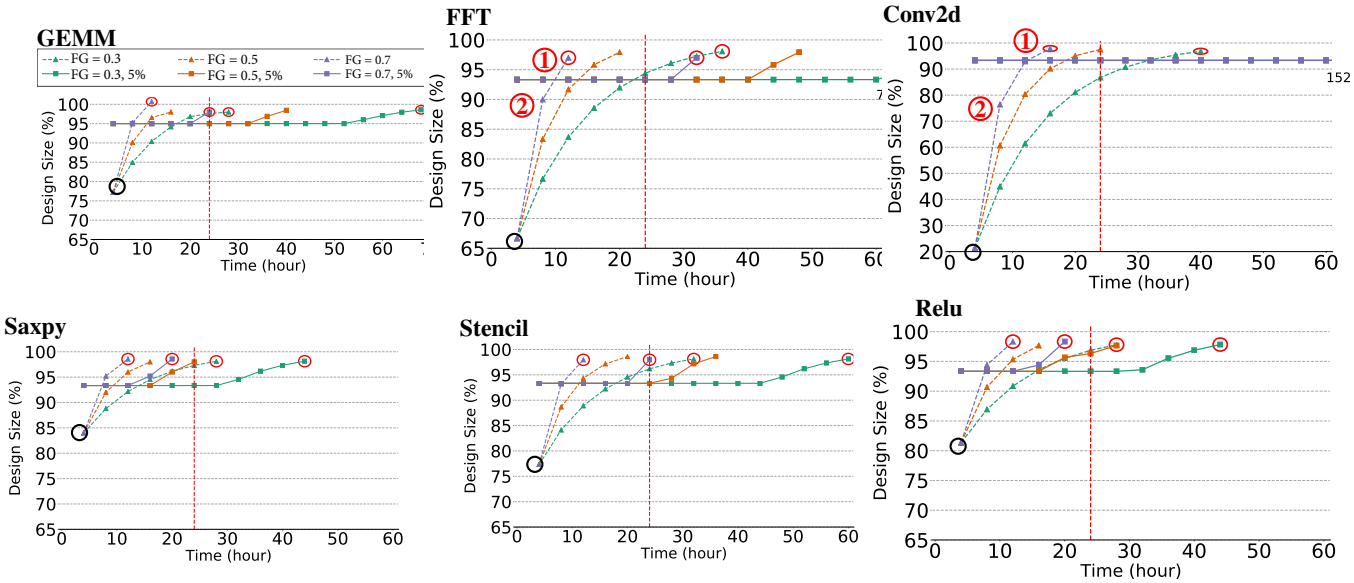


Figure 13: Tradeoffs between different flows in H-RTL checker.

- **1-synth**: We synthesize the H-RTL once onto the FPGA. All guards are built into hardware at the beginning, limiting the logic left-over and consequently H-RTL size.
- **N-synth**: *µgrind* only builds in the required guards in each iteration. As the guard list trims down, we can use the remaining resources for scaling up the H-RTL design size. FPGA bitstream is regenerated for each iteration.
- **N-synth-5%** : We bound the guards to 5% of LUT and BRAM, leaving 95% of the FPGA for the design. More iterations are required, since guards limited per iteration.

Our observations: ❶ The 1-synth approach has to build in the entire guard set at once and this limits the design size that can be checked. For instance, only a CONV design that is 20% the FPGA board could be checked.N-synth can scale up the design size as it trims false guards in each iteration. In the final iteration, N-synth verifies designs that are 5× (CONV) larger than 1-synth. ❷ N-synth exploits the observation that during the checking stage the majority of the H-RTL circuit typically functions correctly. It uses a smaller H-RTL circuit to rapidly eliminate false guards. Within a few iterations (2–3 iterations 9hr) once we have narrowed down the guard's region of interest, we scale up the design-under-test to 90% of the FPGA. ❸ Overall N-synth can complete verification runs on all accelerators in under 24hrs for FG slopes of 0.7 and 0.5. N-synth will complete



X-axis: The timeline in hours. **Y-axis:**. H-RTL size as a % of FPGA resources. ●: 1-synth design size. △ N-synth. ■ : N-synth 5%. Green: FG=0.3 Orange: FG=0.5 Violet: FG=0.7.

Figure 14: 1-synth vs N-synth vs. N-synth-5% (*µgrind* bound to 5% resources). Please view in color and zoom in.

verification between 2×–5 × faster than N-synth-5% as well. N-synth-5% requires extra iterations, since in every iteration it can only check a subset of a large guard-set.

To ensure that the comparison is fair and understand the design space, we introduce deterministic bugs (same site and cycle). We also determinstically control the rate of convergence, using the false guard rate (FG) parameter. For instance, we fix FG rate = 0.7, it means in our model 70% of the guarded signals are trimmed in each iteration until we find the true positive. All models start with the same initial guard set and iterate towards the same bug site. Figure 14 conducts six runs for each accelerator H-RTL: 2 (*N-synth* and *N-synth-5%*) × 3 FG slopes (0.3, 0.5 and 0.7). FP=0.7 is representative of our circuits. The FP=0.3 and 0.5, model complex errors, fewer guards trimmed per iteration.

6.3 Bandwidth requirement (N-synth-BW2%)

We create N-synth-2%BW in which we bound the guards to use 2% of board bandwidth, leaving 98% of bandwidth for the H-RTL circuit. The bandwidth required by the checker depends on the number of guards in each iteration. Thus, we plot the bandwidth required as a % of peak bandwidth in each iteration (see Figure 15). In two workloads, GEMM and Relu, the guards require more than 2% of the bandwidth. Constraining to 2% limit, causes a noteable increase in the number of iterations and time-to-check in those accelerators. In other workloads that guards do not exceed 2% limit and there is no impact on time-to-check.

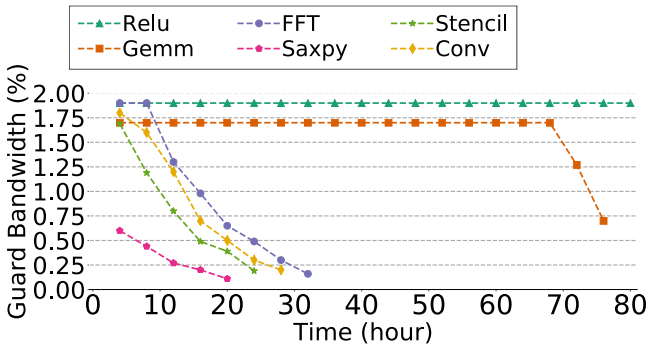


Figure 15: Time-to-check with a hard 2% limit on bandwidth Y-axis: actual bandwidth consumed.

6.4 False-guard rates

Result:*The rate of trimming false guards is high for accelerators as they tend to be parallel and backward slices tend to be shallow. Relu:0.68, Conv2d:0.82, Vadd:0.62, Saxpy:0.61 Stencil:0.78, FFT:0.65.* We plot the $\frac{\#Guarded}{\#Bugs}$, the ratio of the number of netlist signals guarded to the number of actual bugs in the circuit. The number of bugs is fixed, which means this effectively measures the false guards in each iteration. Figure 16 shows how we improve accuracy as we converge on the error. We highlight the rate of change i.e., the higher rate implies that *µgrind* zooms in on the bug faster. The lower the % lower the overhead. We find that the accelerators of 0.69 i.e., 69% of the guarded H-RTL signals will be trimmed in each iteration by the backslice. Note that the rate depends on both the actual circuit dependencies and site of error. These factors are included in the average.

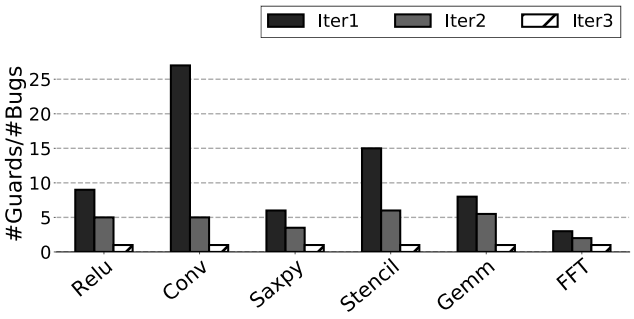


Figure 16: False positive rate: $\frac{\#Guards}{\#Bugs}$ in each iteration. Higher the rate of fall across the iteration, means less false guards.

7 Tool 2: H-RTL Profiler

Result: *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.*

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

In this section, we construct a dynamic profiler for H-RTL circuits. We make the key observation that the number of input bits to a profiler far exceeds the summarized output. Prior state-of-the-art [37] has relied on out-of-circuit analysis. This leads to high DRAM traffic and wastes on-chip SRAM. We realize that profilers tend to be relatively simpler circuits. Embedding profiler circuits in the H-RTL and profiling dynamically during the execution is a better alternative to tracing and profiling.

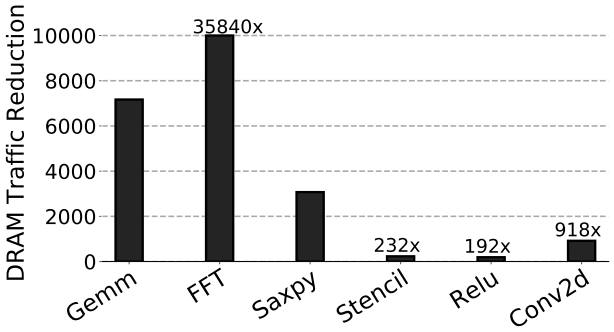


Figure 17: DRAM traffic. *µgrind* vs state-of-the-art [37]

We quantitatively compare *guard*-based profiling against prior state-of-the-art based on tracing. 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 17 shows DRAM traffic reducing in *µgrind* vs state-of-the-art [37] and Figure 18 illustrates *µgrind* SRAM usage vs state-of-the-art [37]. *µgrind* shows promising reduction.

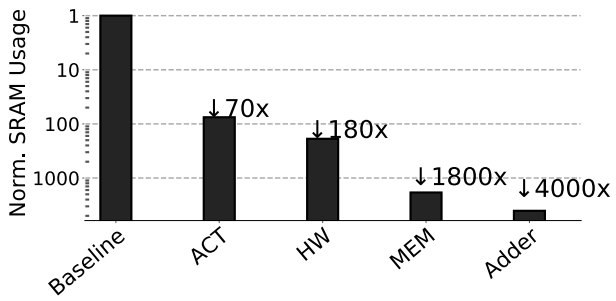


Figure 18: Normalized SRAM.*µgrind* vs state-of-the-art [37]. Lower bar means more reduction in SRAM

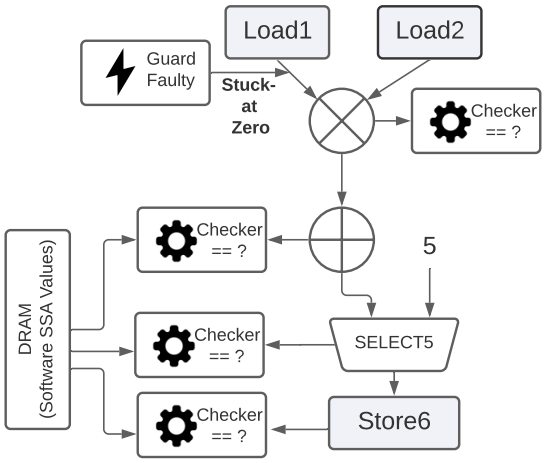


Figure 19: Example of Using guards to inject faults into Load,1 nodes while verifying resiliency in other nodes in the path.

8 Tool 3: H-RTL Faulty

In this section, we study H-RTL circuit resiliency using *µgrind*. We inject two types of guards, faulty and checker. Faulty purposefully injects buggy values into the circuit signals when they are enabled. Checker guards simply check the values against golden values and report if they deviate. This allows faults to propagate unhindered and affect different parts of the circuit. We evaluate how a fault can cause different types of failures. Chiffre [20] provided a convenient framework to inject faults in signals of hand-written RTL. However, Chiffre did not have the ability to check the propagation of faults. It did not permit a user-defined analysis logic, nor vary the instrumentation per signal. *µgrind*'s instrumentation flexibility permits both checker guards and faulty guards to be simultaneously active on the circuit on different signals.

The three categories of bugs we inject are i) **Compute Fault:** We flip the bits and in the ALUs ii) **Control Fault** this case we introduce faults in the branch and merge operators iii) **Memory Fault:** Finally we pertrube the addresses in memory operations and check the impact on the final memory state.

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 20 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 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 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.

9 Hardware Overhead

Table 4 shows the results of synthesis on an FPGA, in terms of the number of LUTs, registers,BRAMs, and Mhz. *µgrind* is a 'pay-what-you-want' approach i.e., there are no fixed overheads. The FPGA resources required depend on how many guards are synthesized and the complexity of these guards. Here we estimate the worse-case overheads of *µgrind* by listing the FPGA synthesis results for H-RTL verifier. We pick the iteration from the earlier iteration since

	LUTs (00s)			Registers (00s)			BRAM (000s)			Mhz		Guard BW	**Worst case**
	Base.	[57]	μ-Veri.	Base.	[57]	μ-Veri.	Base.	[57]	μ-Veri.	Base.	*µgrind*	% Peak BW	Wallclock.
Vadd ($\times U4$)	96	145	127	145	196	175	60	341	78	115	120	7.2%	1.00×
Saxpy ($\times U4$)	67	119	90	109	169	136	34	272	72	112	111	0.2%	1.14×
Conv2D	64	145	84	137	199	165	389	3849	1080	110	108	0.69%	1.00×
Stencil	68	135	76	93	112	105	41	465	82	123	113	0.89%	1.24×
Gemm	217	275	232	111	145	120	142	228	169	121	116	2.1%	1.09×
Relu	79	104	86	101	129	112	485	2667	983	130	122	0.91%	1.64×
FFT	312	390	330	108	194	114	148	8209	329	110	105	0.71%	1.27
Overhead	10—15%			10-25%			1.1×—2.5×			5%		0.7—7%	1.2×

Table 4: AWS F1 Ultrascale+ FPGA Resource and timing. Base: unmodified H-RTL Trace: State-of-the-art HLS [57] μ-Veri.: *µgrind* with maximal guards set up for H-RTL checker; worst case overhead. Exe. time overhead measured with all modules guarded. LUT overhead excludes SAXPY and Vadd since circuit is small evem with unrolling.

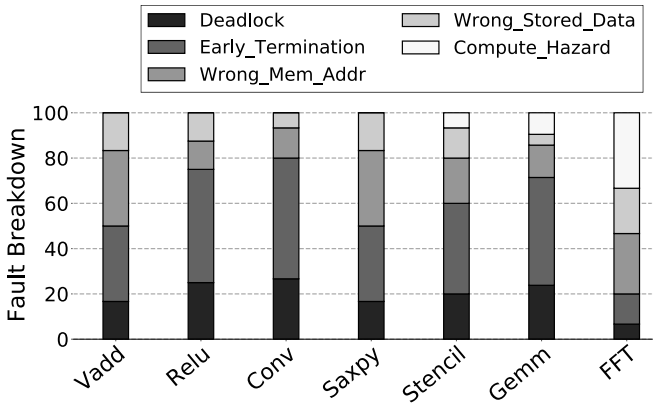


Figure 20: Outcome of Fault injection

they include longer guard lists. Each guard also tends to have higher activity factor and this will require more BRAM values for the shadow buffers in the guards. All in all, this table is a conservative estimate to show the feasibily of *µgrind*. In later iterations in the same verifier we will only require 2% of the FPGA resources. *µgrind* has minimal impact on clock (<5% overhead).

As discussed in the § 3 guards are entirely decoupled from the H-RTL signals, have no dependencies on each other, and do not affect critical path. The logic resources and registers even in the worst-case tend to be limited since the guard analysis functions tend to be simple and require minimal state. The dominant resource overhead is the on-chip BRAMs required to buffer golden values in the verifier (other tools will not include this overhead).

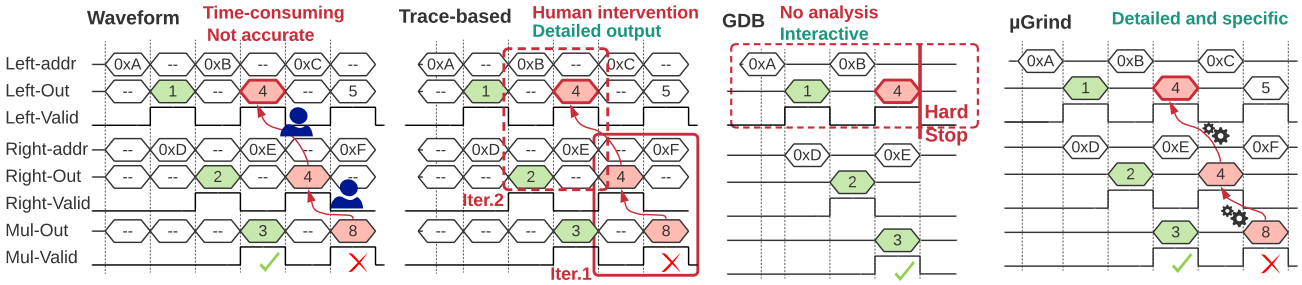
9.1 Related work

Table 5 compares frameworks for H-RTL analysis. We include both state-of-the-art commercial tools e.g., Vivado [12, 55] and state-of-the-art academic tools [26, 29, 31, 32, 57]. Many of the academic tools [32, 57] extend existing commercial tools [16, 55].

The commercial tools (e.g., Xilinx Vivado [55], Legup [15]) support assertions and gdb-like breakpoints. Assertions are "kill-switches" included in the H-RTL at specific signals. Asserts typically check a fixed condition e.g., signal == 0?. They cannot accommodate value-based checks e.g., Is ALU output correct, `ALU.io.out == (io.in1 + io.in2)`?. The RHS in the assert, `io.in1 + io.in2`, has to be dynamically evaluated. Commercial tools lack the support to introduce additional hardware logic for runtime evaluation. Simulations and waveforms [57] in commercial tools can track values using `printfs()`, however all checks are post-execution. The efficacy of asserts also dependen on the user who has to figure out where to insert the assert. Asserts triggers only at deviating signal; the error may have propagated from non-assert location.

In asserts, erring signals could influence each other making it hard to identify the culprit. In contrast, *µgrind* detects the first erring signal and the first cycle deviation occurred. *µgrind* provides a temporal window (neighboring cycles in time) and spatial window (dependent signals) since it patches and lets execution continue. A key novelty of *µgrind* is **Patching**, the ability to modify H-RTL signals during execution. By patching the golden value as soon as a signal deviates, i) we ensure erroneous values do not taint the forward slice, and prevent false reports. ii) we ensure that faulty signals do not influence each other, and we can detect multiple errors simultaneously.

Existing frameworks do not support user-defined tools. They do not permit the user to define the analysis function on the H-RTL signals. They also vary in terms of the **target and type of instrumentation.** Majority of prior tools instrument C/System-C. They require human-in-the-loop to identify the scope [11, 12, 23, 24, 44, 45, 57]. Some of them target hand-written RTL [37], but not verbose HLS-generated RTL. **Execution analysis** In prior work, the analysis of the signals is postponed to post-execution. Thus, signal extraction incurs significant bandwidth penalty. We have demonstrated the benefits of in-execution analysis to save DRAM traffic and on-chip SRAM buffers. **Low-effort**



	Platform	Tgt	Type	Value-based	Asserts	Patch	Low-effort	Auto-edit	Post-Exe Analysis.	In-Exe Analysis	H-RTL size
Source [12, 22, 25]	FPGA	C input	Monitor	—	—	—	—	—	—	—	Large
Monitor [31, 32, 55]	FPGA	C input	Monitor	—	✓	—	—	✓	—	—	Moderate
Asserts [26, 29, 50, 55]	FPGA/ Sim.	RTL	Monitor	—	✓	—	—	—	—	—	Large
Traces [21, 55, 57]	Simulation	C input.	Cause	✓	—	—	—	—	✓	—	Small
Events [37]	Simulation	RTL	Monitor	—	—	—	—	✓	✓	—	Large
CheckPoint [7, 9, 39]	FPGA/Sim.	RTL	Cause	—	—	—	✓	—	✓	—	Small
µgrind	FPGA/Sim.	HLS H-RTL	Root Cause	✓	✓	✓	✓(Iter.)	✓	✓	✓	Large

Table 5: State-of-the-art handwritten RTL and H-RTL Analysis frameworks.

and Auto-edit. The effort required to insert instrumentation into the H-RTL impacts utility. Prior tools require humans intervention to decide where and what to instrument. They lack a flexible mechanism [23, 24, 57] to let a tool determine the instrumentation.

Table 6 illustrated the advantages of *µgrind* vs. Autoslide [57], the state-of-the-art HLS checker. Prior work tackled source-level bugs introduced by the C program fed to the HLS compiler. *µgrind* targets bugs in H-RTL. In Autoslide, the entire analysis phase is post-execution. Hence, they log more than necessary and this leads to a significant bandwidth overhead. Table 4 shows overheads of Autoslide. The largest CONV that *µgrind* can fit on AWS Ultrascale FPGA is R,W,H = 8, 192, 192, and the largest circuit prior work can fit is R,W,H = 8, 56, 56; ≃5× improvement. Autoslide relies on user input to reduce the scope of checking and identifying instrumentation regions. *µgrind* creates a fully automated checker that refines debug scope without user input.

	Prior art [12, 57]	*µgrind*
Monitoring	Offline (post-execution)	Online (in execution)
Checking Region	User [12], Coarse [12, 57], Fine [57]	Whole accelerator (including blackbox RTL)
Region-of-interest	Wide. Buggy and Correct signals	Only dependent signals.
Patching	—	Yes. error does not propagate
Accuracy	User dependent	Convergence guaranteed
Hard failures	Yes.	No. *µgrind* patches values.
Multiple bugs	No. Erring signals propagate.	Guards patch to isolate error.
Design size	Small	Large. See Section 6

Table 6: Tool Comparison: State-of-the-art vs *µgrind* Checker

Advanced debugging and FPGA emulation platforms exist for CPU RTL [39]. They are orthogonal to the problems we target. i) They manually instrument handwritten H-RTL and predominantly support checkpointing and asserts. We target the problem of how to automatically wire user-defined instrumentation into HLS-generated RTL. ii) They also deal with test coverage [36] which is a concern in programmable architectures and dynamic issue RTL such as CPUs. H-RTL is fixed-function and is based on dataflow. Hence, inputs does not dramatically impact circuit coverage concerns.

Commercial formal RTL checkers are only loosely connected to this paper [1]. Logic translation checking is computationally intensive and has only been demonstrated on circuits as complex as floating point ALUs (requires 12hrs [47]). We target the informal problem of finding bugs in complete accelerators (e.g.Convolution) within 24 hrs. Further, SLEC only work with FSMs-with datapath [4] and sequential semantics. Finally, existing tools [6, 17] that profile FPGA performance target an entirely different problem. They track execution time of kernels mapped to the FPGA; at best they are akin to gprof. *µgrind* is the first to profile the activity and values of the internal signals in an accelerator circuit and permit the user to attach custom profilers.

10 Conclusion

µgrind is an open-source framework that enables flexible, low-effort, scalable, dynamic instrumentation of H-RTL. Guards can probe, modify and analyze any H-RTL signal during the execution. Unlike other tools, *µgrind* can dynamically inject values into the H-RTL signal, enabling in-execution tasks such as patching values during verification, and injecting faults during testing. *µgrind* fully automates the process of inserting guard circuits into the H-RTL without requiring any human-effort. *µgrind* is a pay-as-you-go approach where the overheads are proportional to the H-RTL signals monitored. This enables it to verify large circuits that occupy upto 98% of the FPGA.

References

[1] [n.d.]. Catapult High-Level Synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/.

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

[3] [n.d.]. Vivado HLS Co-simulation. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/Vivado-HLS-Co-simulation-Waveform-signals-never-change/m-p/984041.

[4] 2016. High Level Synthesis with a Dataflow Architectural Template. https://arxiv.org/pdf/1606.06451.pdf

[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. 2016. Agile Co-Design for a Reconfigurable Datacenter. In *the 2016 ACM/SIGDA International Symposium*. ACM Press, New York, New York, USA, 15–15.

[6] AMD/Xilinx. 2022. https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded/XSCT-Cross-Triggering-Commands.

[7] Sameh Attia and Vaughn Betz. 2020. StateMover: Combining Simulation and Hardware Execution for Efficient FPGA Debugging. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 175–185.

[8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. [n.d.]. Chisel: Constructing Hardware in a Scala Embedded Language. https://github.com/freechipsproject/chisel3.

[9] Somnath Banerjee and Tushar Gupta. 2012. Efficient online RTL debugging methodology for logic emulation systems. In *Proc. of the IEEE International Conference on VLSI Design*. IEEE.

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

[11] Pavan Kumar Bussa, Jeffrey Goeders, and Steven JE Wilton. 2017. 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)*. IEEE, 1–4.

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

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

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

[15] 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. 2013. From software to accelerators with LegUp high-level synthesis. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 1–9.

[16] Andrew Christopher Canis. 2015. LegUp: Open-Source High-Level Synthesis Research Framework. https://bit.ly/3zttADY

[17] BSC Barcelona Supercomputing Center. 2022. "https://pm.bsc.es/ompss-at-fpga".

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

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

[20] Schuyler Eldridge, Alper Buyuktosunoglu, and Pradip Bose. 2018. Chiffre: A Configurable Hardware Fault Injection Framework for RISC-V Systems. In *2nd Workshop on Computer Architecture Research with RISC-V (CARRV '18)*.

[21] Pietro Fezzardi, Michele Castellana, and Fabrizio Ferrandi. 2015. Trace-based automated logical debugging for high-level synthesis generated circuits. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE, 251–258.

[22] Pietro Fezzardi, Marco Lattuada, and Fabrizio Ferrandi. 2017. 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 (2017), 1–19.

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

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

[25] Jeffrey Goeders and Steven J.E. Wilton. 2017. 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 (2017), 83–96.

[26] Mohamed Ben Hammouda, Philippe Coussy, and Loïc Lagadec. 2014. A design approach to automatically synthesize ansi-c assertions during high-level synthesis of hardware accelerators. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 165–168.

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

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

[29] Yousef Iskander, Cameron Patterson, and Stephen Craven. 2014. High-level abstractions and modular debugging for fpga design validation. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7, 1 (2014), 1–22.

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

[31] Al-Shahna Jamal, Eli Cahill, Jeffrey Goeders, and Steven JE Wilton. 2020. Fast Turnaround HLS Debugging Using Dependency Analysis and Debug Overlays. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13, 1 (2020), 1–26.

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

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

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

[35] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proc. of the FPGA*.

[36] Wisam Kadry, Dimtry Krestyashyn, Arkadiy Morgenshtein, Amir Nahir, Vitali Sokhin, Jin Sung Park, Sung-Boem Park, Wookyeong Jeong, and Jae Cheol Son. 2015. Comparative Study of Test Generation Methods for Simulation Accelerators. In *Proceedings of the 2015 Design, Automation Test in Europe Conference Exhibition* (Grenoble, France) *(DATE '15)*. EDA Consortium, 321–324.

[37] Sagar Karandikar, Albert Ou, Alon Amid, Howard Mao, Randy Katz, Borivoje Nikolić, and Krste Asanović. 2020. FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design. In *Proc. of the ASPLOS*.

[38] 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. 2018. A modular digital VLSI flow for high-productivity SoC design.. In *Proc. of DAC*. ACM Press, New York, New York, USA, 1–6.

[39] Donggyu Kim, Christopher Celio, Sagar Karandikar, David Biancolin, Jonathan Bachrach, and Krste Asanović. 2018. 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)*. IEEE, 76–764.

[40] Youngsik Kim. 2007. *Formal Verification of High-Level Synthesis with Global Code Motions*. Ph.D. Dissertation.

[41] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen. 2019. 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 (2019), 898–911. https://doi.org/10.1109/TCAD.2018.2834439

[42] Chris Lattner. 2020. Many traditional LLVM people are confused about what graph/dataflow semantics means, they've spent a bunch of time working with imperative execution domains. https://bit.ly/3xwcvYs

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

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

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

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

[47] Travis W. Pouarz and Vaibhav Agrawal. 2017. https://s3.amazonaws.com/verificationacademy-news/DVCon2017/Papers/dvcon-2017_efficient-and-exhaustive-floating-point-verification-using-sequential-equivalence-checking_paper.pdf

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

[49] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In *Proc. of the IISWC*. Raleigh, North Carolina.

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

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

[52] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. 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)*. IEEE, 97–108.

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

[54] *μgrind*. 2022. https://anonymous.4open.science/r/d6f70aaf-3014-4353-9b48-cc5759080898/benchmarks/.

[55] Xilinx. 2020. https://bit.ly/39WDXbG.

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

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