Latency Insensitivity Testing for Dataflow HLS Designs

Jianyi Cheng
jianyi.cheng@ed.ac.uk
School of Informatics
The University of Edinburgh
Edinburgh, United Kingdom

Lianghui Wang
lianghuiwang@mail.ustc.edu.cn
Institute of Computing Technology
Chinese Academy of Sciences;
University of Science and Technology
of China, Beijing, China

Zijian Jiang
jiangzijian24s@ict.ac.cn
State Key Lab of Processors
Institute of Computing Technology
Chinese Academy of Sciences;
University of Chinese Academy of
Sciences, Beijing, China

Yungang Bao
baoyg@ict.ac.cn
State Key Lab of Processors
Institute of Computing Technology
Chinese Academy of Sciences;
University of Chinese Academy of
Sciences, Beijing, China

Kan Shi
shikan@ict.ac.cn
State Key Lab of Processors
Institute of Computing Technology
Chinese Academy of Sciences;
University of Chinese Academy of
Sciences, Beijing, China

Abstract

Dataflow high-level synthesis (HLS) tools automatically map a high-level software program to a dataflow hardware design. When testing the design, the HLS tool takes a testing function written in the same software language and translates it into the corresponding hardware testbench for cycle-accurate simulation. However, the generated testbench only follows a fixed schedule to feed inputs and collect outputs, without considering external dynamic behaviors of the test data (also known as latency insensitivity testing). Overlooking these test cases could miss bugs introduced by the designer that occur in certain events, such as late input arrival and output backpressure, leading to higher debugging costs. In this paper, we present an HLS testing toolflow named PEDIA that automatically generates testbenches to detect latency insensitivity bugs. We expand the test space from varying input values to varying input delays. PEDIA automatically determines an efficient test space of input delays based on the HLS schedule for better scalability; and tests the HLS design on a parallel hardware testing platform for high performance. Over two representative sets of existing benchmarks, we show that PEDIA can help identify potential bugs missed by existing approaches and achieves an average speedup of 4.53× on running 1500 test cases.

CCS Concepts

• **Hardware → High-level and register-transfer level synthesis**; **Hardware test**.

Keywords

High-Level Synthesis, Hardware Testing, Emulation, Acceleration

FPGA '25, Monterey, CA, USA

ACM ISBN 979-8-4007-1396-5/25/02
https://doi.org/10.1145/3706628.3708872

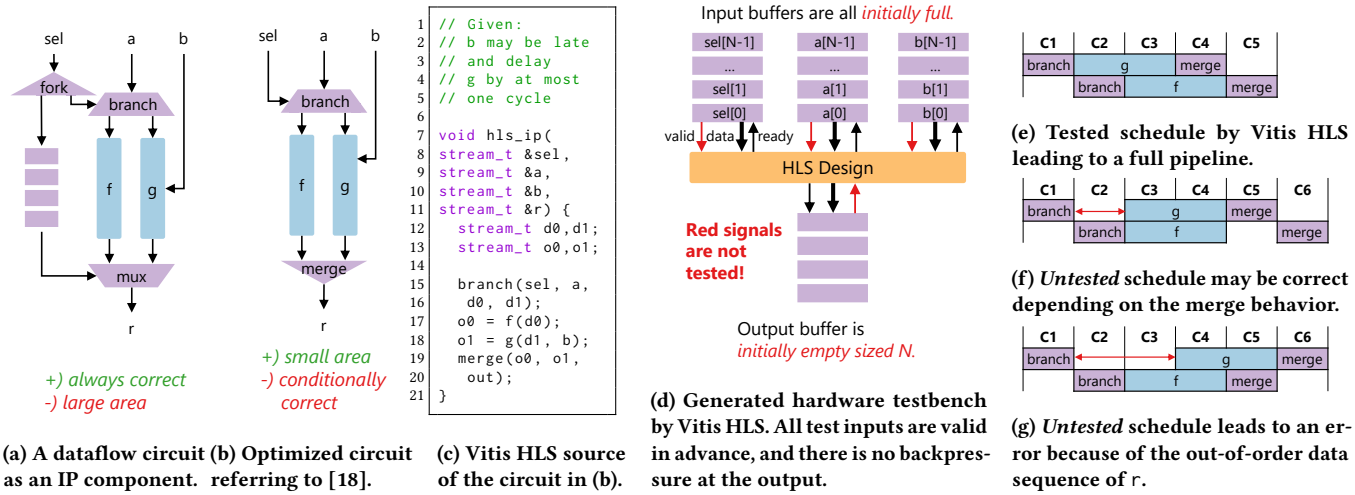
ACM Reference Format:
Jianyi Cheng, Lianghui Wang, Zijian Jiang, Yungang Bao, and Kan Shi. 2025. Latency Insensitivity Testing for Dataflow HLS Designs. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '25), February 27–March 1, 2025, Monterey, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3706628.3708872

1 Introduction

Dataflow high-level synthesis (HLS) tools automatically translate a high-level software program, written in a language like C/C++, into a custom dataflow hardware accelerator, often represented in Verilog/VHDL. They abstract away low-level hardware details from designers, such as instantiating handshake interfaces, scheduling and re-timing, allowing them to focus on architecture exploration with minimal development effort. Many HLS tools today support dataflow HLS, such as through the 'dataflow' directives and the 'hls_stream' library in AMD Xilinx Vitis HLS [1], the front-end library in RapidStream TAPA [20], the 'stream_in' struct in the Intel HLS compiler [26] and the Dynamatic HLS tool from EPFL [28].

Testing is crucial in HLS developments to ensure that the automatically generated hardware meets its specifications and functions correctly. Existing research focuses on HLS translation equivalence [22, 23, 46] but cannot detect design bugs. For example, a designer may introduce bugs by mistakenly writing an incorrect HLS function or directive. To detect such issues, HLS tools automatically generate hardware testbenches from a user-defined testing function, typically written in the same language as the HLS program. The testbench is used to perform cycle-accurate simulations for the generated hardware design. The same HLS program and the user-defined testing function are also simulated as a sequential software program using standard software compilers like GCC, running on identical inputs at the behavioral level. The results of both the hardware and software simulations are then compared to identify any discrepancies, indicating potential design bugs.

However, the above testing method could still miss bugs in dataflow HLS designs. A major limitation is that existing HLS testbenches [1, 28] do not test dynamic data arrivals around the design,



(a) A dataflow circuit as an IP component. (b) Optimized circuit referring to [18].

(c) Vitis HLS source of the circuit in (b).

(d) Generated hardware testbench by Vitis HLS. All test inputs are valid in advance, and there is no backpressure at the output.

(e) Tested schedule by Vitis HLS leading to a full pipeline.

(f) *Untested* schedule may be correct depending on the merge behavior.

(g) *Untested* schedule leads to an error because of the out-of-order data sequence of `r`.

Figure 1: A motivating example of testing an optimized dataflow circuit by the designer. The circuit schedule depends on both the values and arrival times of its inputs, and needs to be both tested. Existing HLS testing only varies the data values in a fixed schedule and leaves other data schedules untested. This may miss bugs such as out-of-order output arrivals.

also known as *latency insensitivity testing*. Instead, they always assume the absence of late input arrival and output backpressure. However, in a dataflow design, a delayed input may lead to a different hardware state. For example, the hardware design may stall part of its operators waiting for a late input to arrive or a backpressure at its output to be resolved at run-time, leading to a dynamic schedule. Although the HLS tool automatically schedules the design to meet the given specifications, designers could still introduce timing bugs.

Verifying a dataflow HLS design and finding potential bugs remain non-trivial tasks. There are two main challenges: 1) HLS language expressiveness and 2) testing scalability. First, both HLS programs and their testbenches are expressed in a *cycle-insensitive programming language*. HLS tools automatically schedule the start times of operations into clock cycles. However, the generated testbench is independent of the HLS schedule, making it difficult to capture cycle-sensitive behavior. Prior work [15, 15, 42] exploits the HLS schedule to accelerate testing but still misses tests on latency insensitivity. Typing systems [17, 37, 38] have also been studied to ensure latency insensitivity of an HLS design, however, they only support designs with fixed latencies. Second, the test space *scales combinatorially* with both input values and their arrival times. The delay of an input could also be infinite, making it impossible to fully explore the test space. The existing testing approach requires designers to manually implement a hardware testbench and restrict the test space using heuristics, by for example using the Universal Verification Methodology (UVM) [21, 25]. In this work, we seek a general and efficient solution to automatically detect latency insensitivity bugs in dataflow HLS designs.

In this paper, we present a testing framework named PEDIA (**P**arrallel **E**mulation of **D**ynamic **I**nput **A**rrivals) for dataflow HLS designs. Given an HLS program and its HLS testbench, PEDIA automatically determines an efficient test space with dynamic test data behaviors and detects latency-insensitivity design bugs. Our main contributions are as follows:

- an end-to-end toolflow that automatically detects latency insensitivity bugs for dataflow HLS designs;
- an automated pass to generate an efficient hardware testing platform on an FPGA that run multiple test cases for an HLS design in parallel;
- a technique that exploits static analysis on the HLS schedule and determines an efficient test space for an HLS design by removing redundant test cases; and
- PEDIA shows better bug detection in latency insensitivity testing compared to existing HLS testbenches, and achieves an average speedup of 4.53× (including FPGA synthesis time) compared to commercial simulators.

2 Motivating Example

We now demonstrate our approach through a motivating example. Figure 1a illustrates a dataflow circuit, where each edge represents a handshake interface. The circuit computes the function `f` or `g` on an input `a` depending on the condition `sel` and returns a result `r`. This is achieved by a branch that sends `a` to one of its outputs and a mux that selects one of the outputs from `f` and `g` to its output, both controlled by `sel`. This design pattern is commonly seen in larger dataflow systems, such as sparse matrix accelerators [28] and neural network accelerators with early exits [11]. For simplicity, `f` and `g` have the same latency, but `g` requires an additional input `b`, which may not arrive at the same time as `a`. The circuit is fully pipelined, so `f` and `g` can be computed in parallel. The latency insensitivity requires `r` to be in the same order as `a`. To ensure this, the fork duplicates the branch condition and feeds it to a FIFO to track the branch choices. This is used to select `r` between the outputs of `f` and `g` to reconstruct the input order of `a`. This implementation works for any data values and arrival times, but could lead to a large area because the size of the FIFO scales with the pipeline depth of `f` and `g`. For example, the FIFO costs 46551 LUTs when `f` is fully pipelined with a latency of 1000 cycles.

Table 1: PEDIA is the first attempt at automated hardware-accelerated latency insensitivity testing for HLS.

	Vitis HLS [1]	Dynamatic [28]	Stratus [3]	BlueCheck [36]	FastSim [5]	FLASH [15]	LightningSim [42]	ENCORE [44]	FireSim [29]	Our work
Platform	CPU	CPU	CPU	CPU	CPU	CPU	CPU	FPGA	FPGA	FPGA
Parallel testing	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓
Hardware acceleration	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
HLS abstraction	✓	✓	✓	✓	✗	✓	✓	✗	✗	✓
Latency insensitivity	✗	✗	✗	✗	✓	✓	✓	✗	✗	✓
Test case reduction	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

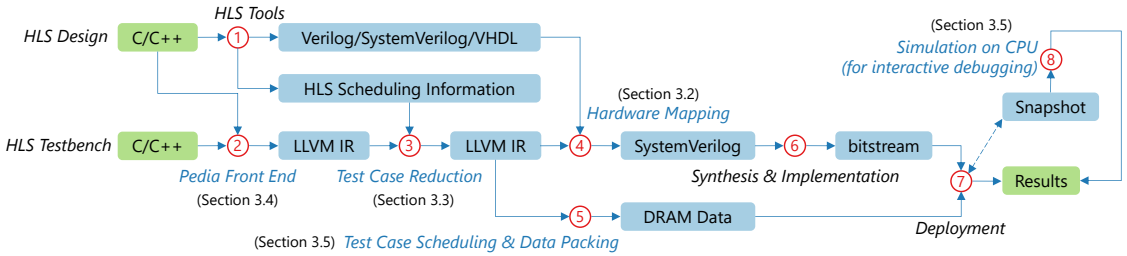


Figure 2: An overview of the PEDIA toolflow. Our contributions are *highlighted*.

An alternative implementation related to the optimization inspired by Elakhras *et al.* [18] is illustrated in Figure 1b and its implementation using Vitis HLS is in Figure 1c. In the optimized circuit, the mux is replaced by a merge, and the FIFO is removed, leading to a smaller area. A merge checks its inputs in a fixed order (always checking `g` before `f` in this case) and sends the first valid input found to its output. The circuit may still be correct under constrained input values and arrival times. For example, assume `b` may be late and delay `g` by at most one cycle. This is common when loading `b` from the off-chip memory or an external circuit that gets stalled. The computation is still correct as illustrated in Figure 1e and Figure 1f. However, further delaying `b` could cause errors, such as `r` being out of order in Figure 1g.

It is designers' responsibility to verify their HLS design's correctness under their target use cases. Designers often rely on hardware simulations to detect design bugs. Existing HLS tools, such as Vitis HLS [1] and Dynamatic [28], generate a testbench as shown in Figure 1d. The HLS design is the device under test (DUT), and connected to three input buffers and an output buffer, both sized sufficiently to hold all the data. The test inputs are all initialized in these input buffers. This leads to a fully pipelined schedule of DUT, also known as the 'best-case schedule', as shown in Figure 1e. However, other schedules such as those in Figure 1f and Figure 1g cannot be tested because the input and output buffers are always ready to feed and consume data, respectively.

The absence of these test cases prevents the designer from verifying customized optimizations on their HLS hardware. For example, Figure 1f verifies whether the merge is checking its inputs in the right order, otherwise the results may be wrong. The designer may also input wrong specifications for their use cases, leading to an unexpected schedule, as in Figure 1g. In order to capture such design bugs, PEDIA automatically generates a custom testing platform for an HLS design and tests with dynamic data arrivals. Both the input values and arrival times affect the behavior of the DUT, leading to a combinatorially growing test space. This makes existing HLS testing that sequentially runs test cases on a CPU no longer scalable. PEDIA addresses this by mapping test cases to an FPGA-based hardware testing platform, leveraging spatial data parallelism among similar test cases for high performance.

Problem Formalization

A key novelty of our work is that PEDIA expands the existing HLS test space from data values to data schedules. Let D be the set of all possible test inputs and T be the set of all possible schedules for an HLS design. The complete test space is then $D \times T$. Given user-defined inputs $D' \subseteq D$, existing HLS testbenches test $D' \times T'$, where $T' \subseteq T$ are the schedules without late input and output backpressure. PEDIA generates a testbench with a test space $D' \times T''$, where $T' \subseteq T'' \subseteq T$. In the rest of the paper, we show how to determine an efficient T'' and explore it in parallel.

3 Methodology

PEDIA is an independent toolflow on its own hardware testing platform for general HLS designs including non-dataflow ones. When testing a non-dataflow HLS design, PEDIA works the same as existing HLS testbenches because the HLS design follows a static schedule. Similar to other HLS testing frameworks [1, 28], PEDIA requires both the HLS program and the corresponding testbench as its input. The front end of PEDIA reuses existing Vitis HLS testbench syntax to save effort in rewriting existing HLS tests. Figure 2 illustrates a high-level overview of the PEDIA toolflow.

① The traditional synthesis flow of an HLS tool takes an HLS program and automatically generates a functionally equivalent hardware design. The HLS tool also generates a scheduling report that describes the hardware states of the HLS design.

② PEDIA accepts inputs in C, C++, and other programming languages that can be translated into LLVM IR. Here, we parse the

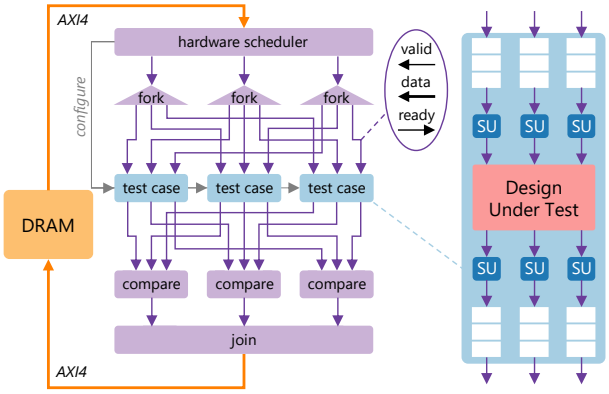
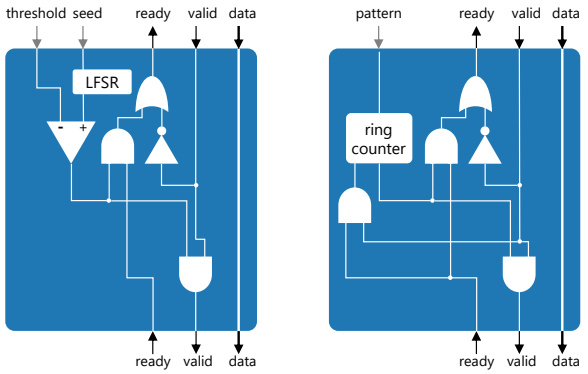


Figure 3: A hardware platform for testing a 3-input and 3-output HLS design in parallel. SU = stalling unit.



(a) Probability-based mode **(b) Pattern-based mode**

Figure 4: Stalling units (SUs) configured in two modes.

C / C++ input using Clang [35] for prototyping. Designers can optionally add cycle-sensitive constraints in custom pragmas to test special use cases, described in Section 3.3.

③ A test space is constructed from the PEDIA testbench, and further optimized by static analysis on the HLS schedule to remove redundant test cases, described in Section 3.2.

④ A custom hardware testing platform is generated to run multiple test cases in parallel, described in Section 3.1.

⑤ PEDIA schedules both test cases for efficient acceleration, fully exploiting the available resources, described in Section 3.2.

⑥-⑦ The testing platform is deployed on an FPGA and computes like a hardware accelerator.

⑧ The intermediate states during hardware testing on an FPGA are not directly accessible. PEDIA provides an interactive debugging interface through the snapshot of the testing platform, described in Section 3.4.

3.1 Hardware Testing Platform

We now describe our hardware testing platform for latency insensitivity testing. Given a combinatorially growing test space, we choose to test on a hardware-accelerated platform in order to run test cases at high performance and in parallel. The architecture of the hardware platform faces two design choices. First, it could follow the same architecture in Figure 1d, where all the test data is stored on-chip and fed into the DUT. However, this may not be amenable for designs that require a large memory size, such as systolic arrays. Also, the scheduler for each test case needs to be customized, leading to a scalability challenge. Second, the testing platform could also be a large dataflow system with the DUT in the loop. This architecture lifts the restriction on on-chip memory size by streaming and can be generalized to all test cases by varying component configurations at run time. We choose the latter to maximize the generality our approach.

The main design goal of the proposed platform is that the testing process can scale efficiently to thousands of test cases. First, the general platform architecture accepts arbitrary HLS designs, only handling their interfaces such as BRAM and handshake. Second, the simulated delay must be software reconfigurable between test cases in the same hardware architecture, leading to a short reconfiguration time and avoiding repeated FPGA synthesis. Finally, the control granularity on the simulated delay can target both average and cycle-accurate behavior.

3.1.1 Stalling Units. PEDIA maps the simulation of the test cases onto an FPGA using a custom hardware platform for acceleration, as illustrated in Figure 3. The right of the figure shows a single test case, mapped into a test instance. A test instance contains a copy of the HLS design as the DUT, connected its inputs and outputs to *stalling units (SU)*. An SU does not change data values, but blocks them to simulate delay events, such as late input arrival or output backpressure. The delay behavior of each SU is reconfigured by the hardware scheduler at the beginning of each iteration.

The detailed implementation of an SU is demonstrated in Figure 4. A single SU can be configured in two modes, where we show their circuit designs separately for simplicity. First, the *probability-based* SU, shown in Figure 4a, generates pipeline stalls based on a probability and a seed, set to 0.5 and 0 by default, but both could be customized by the designer. The delay behavior is determined by a linear feedback shift register (LFSR) and a constant threshold determined by the given probability. The generated number below the threshold leads to the data being blocked in the next clock cycle. This simulates pipeline stalls with a probability p, approximated by the following, where b is the bitwidth of the LFSR register.

$$p \approx \frac{threshold}{2^b - 1} \tag{1}$$

Figure 4b shows the *pattern-based* SU, which generates pipeline stalls following a user-defined schedule. The schedule is stored in a bit vector, and each bit indicates an SU state in a specific cycle. A ring counter enumerates these bits periodically and blocks the data in the next clock cycle if the current bit is 0. In comparison, the probability-based SU provides uncertainty, enabling designers to fuzz on different test cases to detect potential bugs; and the pattern-based SU allows designers to have cycle-accurate controls of the test case and perform verification on a specific test case.

3.1.2 Test Case Parallelism and Isolation. At the top level on the left of Figure 3, the test data is stored in DRAM for better scalability, and

accessed via the AXI interface at run-time. The hardware resources of an FPGA are often sufficient to run multiple test cases in parallel, where each test instance checks a specific test case. For example, three test instances are running in parallel in Figure 3. All the purple edges are handshake interfaces, making the testing platform a large dataflow hardware accelerator. A *hardware scheduler* initiates the process by fetching testing configurations and applying them to SUs to simulate data delays. In each iteration, these test instances run on the same test data but have different SU configurations to simulate different test cases. Sharing the same data reduces the available memory bandwidth restriction while preserving high parallelism. The test data is replicated and distributed to each test instance using the *fork* component. The output values from different instances are expected to be identical. They are *compared* using hardware comparators. Any mismatch will return a fault signal to the system, indicating an error. An iteration finishes when the slowest instance finishes, synchronized by the join component. The *join* component also sends a copy of the results to the host for comparison with the golden reference. The testing iterations are carried out sequentially.

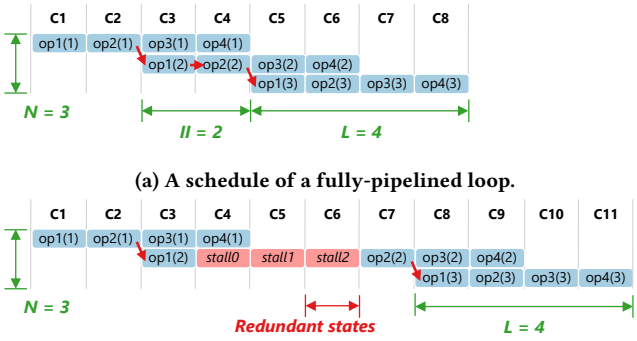
A key design constraint of the testing platform is that the computation of these test instances must be independent and all the stalls must be controlled by the SUs. For example, a test instance may send backpressure to the fork, which stalls other instances at the outputs of the fork, leading to a mismatch between actual data arrival and test constraints. To avoid interference between parallel test cases, *buffers* are inserted around the SUs, as shown in the right of Figure 3, to balance the throughput between the external components and the SUs. A sufficient buffer depth must be determined to ensure the absence of backpressure. A large buffer may lead to low area efficiency, while a small buffer may cause mismatched test results. PEDIA applies static analysis on the HLS schedule and sets the buffer size for an interface to $2d$, where d satisfies both the following constraints.

$$\frac{N-d}{\theta_{\text{ext}}} \leq \frac{N}{\max \theta_{\text{hls}}} \tag{2}$$

$$\frac{N-d}{\min \theta_{\text{hls}}} \leq \frac{N}{\theta_{\text{ext}}} \tag{3}$$

Constraints 2 and 3 describe buffering and prefetching, respectively. N is the total value count for a test case. θ_{ext} is the throughput of an external component, such as a fork or a compare in Figure 3. θ_{hls} is the throughput of the HLS interface. For an HLS design with dynamic throughput, PEDIA takes the minimum and maximum throughput values from the HLS scheduling report for the above analysis. The interface throughput also depends on the delays in its SU. We also focus on the maximum and minimum delays of an SU among all test cases for estimating θ_{hls}. Still, d only reflects the average case, so we approximate the buffer size to $2d$ and add hardware checks at each buffer interface to detect any unexpected stalls at run time, leading to an invalid flag at the end of the test. In our experiments, we observed that $2d$ is sufficient to balance the throughput over all the benchmarks.

The combination of configurations among many SUs opens up a large test space, covering diverse input arrivals and output backpressure. However, the test space size grows combinatorially due to the increasing number of potential delays, making a naive exhaustive search approach impractical. In the next section, we will describe how static analysis of the HLS hardware design can be exploited to efficiently reduce the test space.



(a) A schedule of a fully-pipelined loop.

(b) A schedule of the pipelined loop with pipeline stalls.

Figure 5: The schedule of a pipelined loop in an HLS design. The red arrows represent the dependence between two operations, and the red bubbles represent run-time pipeline stalls. N = loop trip count. *II* = initiation interval. L = iteration latency. The stalls in (b) lead to additional states in C4 and C5 to test, but further adding delay does not add new states.

3.2 Test Case Reduction and Scheduling

3.2.1 Test Case Reduction. We now demonstrate how to restrict the set of SU configurations to reduce the test space. PEDIA applies static analysis on the HLS design's schedule. The schedule of an HLS design contains all possible hardware states of the hardware design in clock cycles. A dynamically schedule design may contain input-dependent hardware behaviors, and PEDIA focuses on the one with maximum latency, also known as the 'worst-case' schedule.

For example, Figure 5a illustrates the schedule of a fully pipelined loop. A fully pipelined loop schedule typically includes three constraints. First, the initiation interval *II* represents the number of clock cycles between consecutive loop iterations. Second, the iteration latency L represents the latency of a single-loop iteration. Finally, the loop trip count N represents the total number of loop iterations. While these constraints can vary at run-time, here we assume they remain constant for simplicity.

Figure 5b presents a schedule of the same loop with run-time pipeline stalls. For example, the operation `op2` in the second iteration has a pipeline stall for three clock cycles, waiting for `op1` to load data from shared memory. These pipeline stalls lead to additional hardware states, as shown in cycle **C4** and **C5** in the figure. These states are crucial for testing the ability of the HLS design to handle backpressure. However, longer stalls might not add new states if all independent operations are completed and the HLS design is only waiting for the input. Such scenarios result in redundant states in testing, such as cycle **C6** in the figure.

These redundant states unnecessarily increase the test space, because endless pipeline stalls lead to an infinite number of schedules to test. To effectively reduce the test space while preserving effective coverage of hardware states, PEDIA analyzes the HLS schedule and restricts the maximum delay latency for each SU, where the maximum delay latency is the maximum pipeline depth

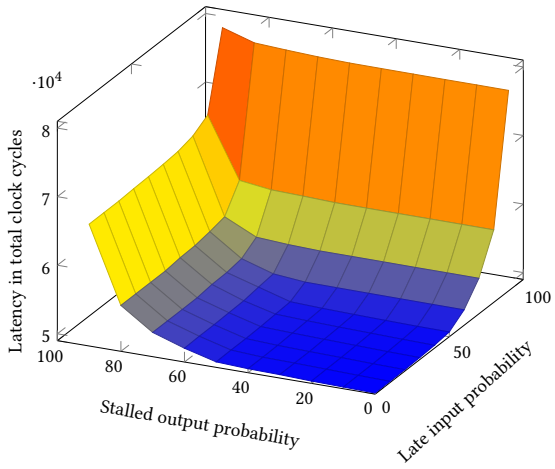


Figure 6: The test latency of benchmark `MM` depends on the pipeline stalls in its surroundings. For a large set of test cases, PEDIA automatically groups those with similar latencies to run in parallel, avoiding long synchronization time in the join component (Figure 3).

of the HLS design. For each probability-based SU, the hardware scheduler counts the cycles where no SU passes any token and forces the data to pass when the maximum delay latency is reached; and for each pattern-based SU, no optimization is required since the pattern is always a finite-length bit vector provided by the designer.

3.2.2 Test Case Scheduling. The restriction on SU behaviors for reducing the test space also leads to a simpler hardware design of the testing platform. The testing platform can be treated as an FPGA accelerator, so traditional FPGA design optimizations can be applied to improve area efficiency and performance. Here we explain four key optimizations used in this work. All these optimizations are general and applied to accommodate all test cases.

1) Data Parallelism maximizes the number of parallel test cases within the available hardware resources. PEDIA obtains the area of an HLS design from the synthesis report and determines efficient parallelism in a greedy form. The memory bandwidth does not scale with the number of parallelism, because the test cases in the same iteration run on the same test data.

2) Data Packing packs multiple values in a single memory line and accesses them in a single clock cycle, improving memory density. PEDIA statically determines a data packing scheme by checking the variable bitwidth and their throughput from the HLS schedule. This data packing scheme does not change throughout the testing process, because the data format does not change among test cases. PEDIA also programs the corresponding decoding logic for the hardware scheduler in Figure 3.

3) Data Prefetching prefetches data before testing, when the feeding throughput of the fork in Figure 3 is lower than the consuming throughput of the HLS design. The prefetching depth is statically determined by the buffering constraints in Constraints 2 and 3.

4) Test Case Scheduling maps test cases with similar latencies in the same iteration. This reduces the synchronization time of the join component, because the latency of an iteration depends on its slowest test case. PEDIA estimates the latency of each test case based on its SU configurations. Figure 6 shows that more frequent delays in the SU lead to longer total latency.

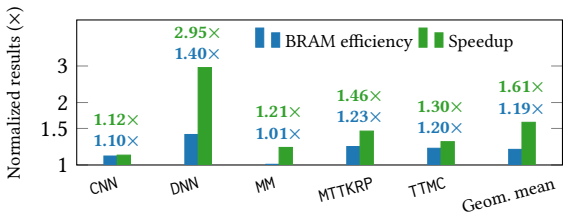


Figure 7: PEDIA applies static analysis on the HLS schedule and optimizes the hardware testing system for higher area efficiency and lower testing latency. This figure shows the normalized results of the static analysis-optimized hardware testing compared to the one without optimization.

In each iteration, the hardware scheduler reconfigures the SUs for all the test cases, prefetch the test data and start all the test instances. The hardware optimizations significantly improve area efficiency in BRAM usage and speedup testing time, as shown in Figure 7. Compared to the vanilla testing platform, the optimized system achieves up to 1.4× BRAM efficiency and 2.9× speedup. The better area efficiency is due to optimizing buffer size by restricting the maximum delays of SUs. The delay restriction in SUs also improves the latency, as well as the optimizations above.

3.3 Use Case-Directed Testing

For a given HLS testbench, PEDIA automatically sweeps three probabilities by default for stalling each data interface, 0.1, 0.5 and 0.9, leading to a set of test cases. However, designers can also customize these constraints for their test data. This is particularly useful when the HLS design is optimized for special use cases, such that the HLS tests can ignore unrealistic data arrival times and focus on a pre-defined test space. Existing HLS testbenches do not provide such a user interface for designers because the HLS languages are cycle-insensitive. Designers have to manually implement their own testbenches for use case-directed testing. In order to lift this restriction, PEDIA provides an optional cycle-sensitive interface for designers to customize their tests.

For example, Figure 8 shows an example of PEDIA testbench for the HLS design in Figure 1c. The original source of the testbench remains the same as Vitis HLS. The injection of pipeline stalls can be expressed using two types of pragmas. First, a pragma can set pipeline stalls for a data interface based on a given probability. For example, the pragma in line 8 states that every clock cycle `sel` has a half chance of being stalled. This is suitable for large-scale fuzzing of the HLS design to explore various timing behaviors of inputs and outputs. Second, a pragma can also impose a static schedule on a data interface, offering cycle-accurate control of its arrival. For example, the pragma in line 10 sets the ready signal of `r` to a static schedule with a period of five clock cycles. These pragmas lead to probability-based and pattern-based SU insertion in hardware respectively, as described in Section 3.1. There are

```
1  // PEDIA HLS test bench
2  int main() {
3    ...
4    // Start hardware testing
5    for (int t = 0; t < 3; t++) {
6    // There is a 50% chance that sel gets
7    // blocked every cycle.
8  #pragma PEDIA port=sel probability=0.5
9    // r has a backpressure every five cycles.
10 #pragma PEDIA port=r pattern="11110"
11     vectmult(sel[t], a[t], b[t], r[t]);
12   }
13   ...
14 }
```

Figure 8: PEDIA testbench for the HLS program in Figure 1c. PEDIA reuses existing HLS testbench source and allows designers to input *cycle-sensitive* constraints for testing on special use cases, such as 'adding delays based on a probability (line 8)' or 'adding delays following a given pattern (line 10)'.

```
1  // Scalar arguments in
2  // Dynamatic are handshaked.
3  int top(int a,
4  int b[N],
5  stream_t &c) {
6  // In the HLS testbench:
7  // #pragma PEDIA port=a
       probability=0.5
8  // #pragma PEDIA port=c
       probability=0.5
9    auto d0 = f(a);
10   auto d1 = b[3];
11 #pragma PEDIA port=d0
       probability=0.5
12 #pragma PEDIA port=d1
       probability=0.5
13   c = d0*d1;
14   return c;
15 }
```

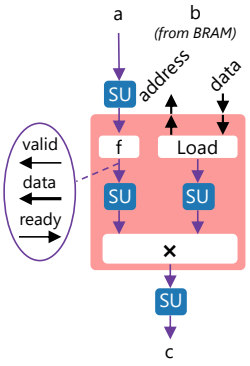


Figure 9: An example of PEDIA injecting SUs both inside and outside an HLS design in Dynamatic.

also other pragmas for full control of the test, such as customizing the buffer size of a particular interface and initial seed for the probability-based SU. These pragmas are for annotation only and can be applied onto diverse HLS tools and their testbenches.

3.4 Practical Testing and Design Considerations

In this section, we discuss practical needs beyond HLS verification and describe our contributions in the debugging phase. First, the adaptive interface of the SU enables seamless integration into any handshake interface. PEDIA can also insert an SU into an existing handshake interface *inside a dataflow HLS design*. This helps a designer identify bugs when a bug is detected at the top level. For HLS tools that contain abstractions describing the hardware dataflow architecture, such as Dynamatic [28] and TAPA [20], PEDIA analyzes the abstraction and inserts SUs into the design for fine-grained testing. Taking Dynamatic for example, Figure 9 shows the source of an HLS design and its test instance mapped into the testing platform. On the left of the figure, the data behavior of its surroundings is specified in the HLS testbench (lines 7-8), and the internal data delays are specified in the HLS source using the same pragma formats (lines 11-12). PEDIA inserts SUs into these handshake interfaces on the right to achieve fine-grained testing. However, the number of SUs inside the DUT could be large leading to a significant drop in clock frequency if they are all controlled by the hardware scheduler. Instead, PEDIA hardens the configurations for SUs inside the HLS design, and only dynamically reconfigures the external SUs using the hardware scheduler in Figure 3.

Second, the intermediate states are not accessible by the designer when running on the PEDIA hardware testing platform. This may increase debugging effort when a bug is detected, such as deadlock. In order to address this, PEDIA orchestrates the snapshot feature provided by a hardware emulation framework named ENCORE [44] for *interactive debugging*. ENCORE is a framework that exports snapshot of an FPGA design to the host at run-time. It does not require modification of DUT so the integration can be automated. PEDIA passes our testing platform as the whole DUT in ENCORE and samples snapshots at a constant time interval. We made modifications on the ENCORE overlay, but the HLS DUT does not need to be modified, leading to automated generation of a debugging platform. During the testing process, an FPGA snapshot of the testing platform is exported after a user-defined period, including all the signal states at run time. The snapshot can be passed to a hardware simulator, such as Xilinx Vivado XSIM [2] and Siemens ModelSim [4], to continue testing using traditional hardware simulation. This allows users to 'teleport' to a certain breakpoint through hardware acceleration and observe the run-time signals for debugging.

Finally, a DUT may require a higher memory bandwidth than the available off-chip memory bandwidth on the FPGA device for testing. Prefetching data may work but require a large buffer size, potentially leading to a hardware mapping failure. PEDIA provides a pragma to add clock divider to the HLS design and slows it down by a user-defined factor. This allows the DUT bandwidth to match the off-chip memory bandwidth, at the cost of long testing latency for each test case.

There are two limitations of the PEDIA testing framework. First, a large HLS design may not fit on the target FPGA, leading to an error raised by PEDIA. This requires the designer's effort to partition the HLS design for effective testing. This does not miss latency-insensitivity bugs because the SUs effectively simulate neighbor behaviors for each partition. Second, accurate coverage analysis on latency insensitivity remains a research challenge. The existing effort on coverage analysis focuses on the register transfer level (RTL) level [33]. PEDIA is the first attempt at latency insensitivity testing for dataflow HLS designs, and HLS coverage analysis will be our future work.

4 Experiments

We evaluated PEDIA on several dataflow HLS benchmarks. For testing capability analysis in bug detection, we compared PEDIA with HLS testbenches generated by Vitis HLS and Dynamatic. For performance analysis in testing time, we compared PEDIA with commercial simulators, Xilinx XSIM [2] and Siemens ModelSim [4]. To ensure fairness in performance evaluation, we manually added the same dynamic input behaviors as PEDIA to the HLS testbenches for simulation. We also evaluated the area overhead of the proposed hardware testing platform. We obtained the total clock cycles from

Table 2: Overall hardware results of the HLS design over two benchmark sets and their testing platforms generated by PEDIA. The upper part of the table have coarse-grained architecture and are tested with SUs around the HLS designs, and the lower part of the table have fine-grained architecture and are tested with SUs inside the HLS designs. All SU probability = 0.5.

Benchmarks	Standalone HLS design						PEDIA testing system									Normalized Latency per test (×)
	LUTs	DSPs	BRAM	Fmax (MHz)	Total cycles	Latency (μs)	Parallelism	LUTs	DSPs	BRAM	Fmax (MHz)	Total cycles	Latency (μs)	Latency per test (μs)	Synthesis time (s)	
CNN	195k	1.51k	417	295	39.2k	133	1	195k	1.51k	526	211	68.1k	323	323	1715	0.41×
DNN	117k	192	480	334	15.6k	46.8	2	233k	384	615	210	32.5k	155	155	2046	0.30×
MM	92.1k	1.73k	270	292	7.46k	25.6	1	96k	1.73k	270	212	10.1k	47	47	1577	0.54×
MTTKRP	138k	1.8k	273	308	72.7k	236	1	144k	1.8k	673	212	105k	497	497	1683	0.47×
TTMC	163k	1.56k	373	303	347k	1144	1	166k	1.56k	656	213	593k	2.79k	2.79k	1647	0.41×
histogram	713	0	0	359	1012	2.82	200	262k	0	600	200	16443	82	0.411	14077	**6.8×**
fir	630	3	0	353	1.02k	2.87	450	396	1.35k	900	163	15.3k	94.6	0.210	5403	**13.7×**
iir	1.00k	6	0	343	5k	14.6	150	237k	900	300	185	18.5k	100	0.668	1944	**21.8×**
image_resize	1.05k	0	0	341	941	2.76	150	260k	0	150	201	17.3k	86.2	0.574	2972	**4.80×**
insertion_sort	1.67k	0	0	331	526	1.59	80	226k	0	80	182	8.37k	46.0	0.574	2350	**2.77×**
jacobi_1d	1.18k	3	0	345	1.18k	3.41	150	282k	450	300	201	8.18k	40.8	0.272	1567	**12.6×**
video_filter	2.45k	9	0	335	945	2.82	80	273k	720	240	197	18.6k	94.6	1.18	3342	**2.39×**

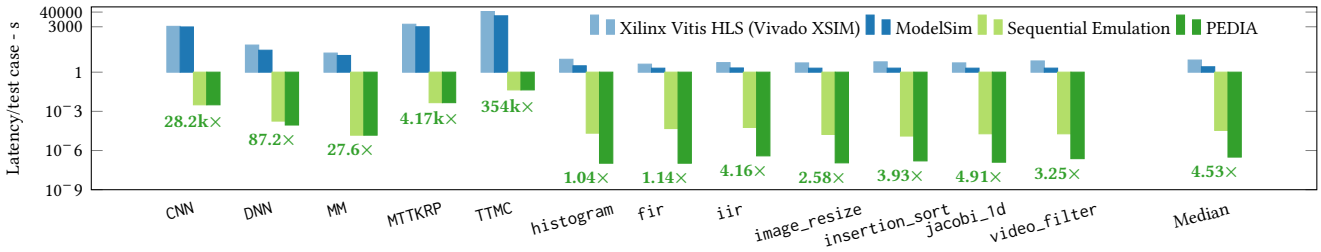


Figure 10: Comparison of the testing time (without synthesis time) between PEDIA and commercial tools on running 1500 test cases. The speedup annotates the normalized testing speed of PEDIA (including synthesis time) compared to Vivado XSIM.

the Vivado XSIM simulator and the area results from the Post Place & Route report in Vivado. The FPGA family we used for evaluation is Xilinx Zynq UltraScale+. The version of Xilinx software is 2020.2, and the version of ModelSim is 2020.4.

4.1 Benchmarks

There is no existing benchmark set for HLS design bugs or vulnerabilities. Here we illustrate three artificial but commonly used examples including the motivating example, where existing HLS testbenches potentially miss bugs. To demonstrate PEDIA's broad applicability to various HLS tools, we chose two distinct HLS benchmark sets from Vitis HLS-based AutoSA [48] and Dynamatic [28], each generating designs with different dataflow architectures.

AutoSA contains five distinct benchmarks: `CNN` is a convolutional kernel from a convolutional neural network; `DNN` is a fully connected layer from a deep neural network; `MM` is a general matrix multiply function; `MTTKRP` is a matricized tensor times khatri-rao product function; and `TTMC` is a tensor times matrix-chain function.

We also choose seven realistic benchmarks from Dynamatic [28] for evaluation: `histogram` sums various weights onto the corresponding features in a sparse form; `fir` is a finite impulse response filter function; `iir` is an infinite impulse response filter function; `image_resize` resizes an image based on a given offset; `insertion_sort` is an insertion sort function; `jacobi_1d` is a 1-D Jacobi stencil function; and `video_filter` is a filter function for videos with frames in RGB values. Both benchmark sets are publicly accessible. We reproduced the HLS designs from these benchmarks and used PEDIA for parallel HLS testing.

4.2 Testing Capability

Here we present two case studies on how PEDIA detects latency-insensitivity bugs. The first example in Figure 11a shows a potentially missed bug in a dataflow multiplier by the Vitis HLS testbench. Vitis HLS uses 'hls_stream' (annotated as `stream_t`) to instantiate a handshake interface. Such a handshake interface provides non-blocking APIs, `read_nb` and `write_nb`, where a load or store still computes with absence or excessive copies of data. These APIs are useful for dynamic dataflow behaviors, but may cause latency insensitivity bugs. For example, late arrival of `b` or backpressure from `c` may lead to a non-deterministic behavior. Existing solution suggested by the official documentation of Vitis HLS [39] is to manually implement a separate RTL testbench from the HLS testbench for exhaustive testing. PEDIA addresses this by automatically generating a customized testbench for a given HLS design to perform such latency insensitivity testing.

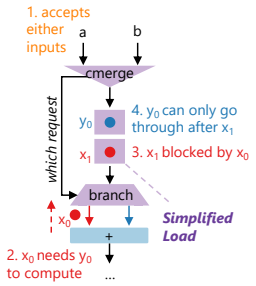
The second example in Figure 11b shows a working HLS design in Dynamatic under specific constraints but cannot be correctly verified by the existing testbench. The design adds two elements from the same array and returns the result. This may lead to deadlock if the merge selects an inappropriate order to serialize `a` and

```
1  void mult(
2  stream_t &a,
3  stream_t &b,
4  stream_t &c) {
5    TYPE d0, d1;
6    if (a.empty())
7      return;
8    // not checking b
9    d0 = a.read_nb();
10   d1 = b.read_nb();
11   // not checking c
12   c.write_nb(d0*d1);
13 }
```

(a) Non-deterministic dataflow behavior in Vitis HLS [1]

```
1  int add(
2  int a,
3  int b,
4  int d[N]) {
5
6    int x,y,r;
7
8    x = d[a];
9    y = d[b];
10
11   r = x+y;
12   return r;
13 }
```

(b) A load adder in Dynamatic [28]



(c) Deadlock state in (b).

Figure 11: Two case studies of how known issues in different HLS tools can be tested in PEDIA.

b. For example, Figure 11c shows a deadlock state. For simplicity, a shift register is used to model the pipeline of the load operation. 1) The example shows one possible execution trace where `x = d[a]` executes twice before `y = d[b]` executes, because the merge always chooses a first when a and b are both valid. 2) The adder receives the first x and needs to pair it with the first y to process. 3) The second x at the output of load is blocked by the backpressure sent by the adder because the data path already holds the first x. 4) The first y is queued in the pipeline, waiting for the second x to be released to reach the output. This is the schedule when a and b are both valid. The circuit still works correctly for the use case where a and b always arrive alternatively. The Dynamatic testbench cannot accept such cycle-sensitive constraints to test special use cases of an HLS desigm, while these constraints can be accepted and tested on the HLS design by PEDIA.

4.3 Hardware Testing Platform Area

The overall hardware results of the tested HLS designs and their corresponding testing platforms are shown in Table 2. Compared to sequential testing on an FPGA, the parallelism enabled by PEDIA demonstrates a significant speedup. The upper part of the table generates HLS designs with larger areas, because the optimizer in AutoSA tries to utilize maximum hardware resources on the target FPGA for high performance. Most of these designs are typically limited by the available DSP resources and cannot be duplicated for parallel testing. PEDIA then performs sequential testing with SUs. The area overhead of the hardware platform is relatively small, up to 4% the area of the HLS design. The integration of SUs increases the critical path, leading to a lower clock frequency.

The lower part of Table 2 generates smaller HLS designs compared to the upper part. These designs have a fine-grained dataflow architecture consisting of several dataflow components connected via handshake interfaces. For these benchmarks, PEDIA integrates SUs inside an HLS design and replicates the HLS design extensively for parallel testing. This approach has achieved up to 450 concurrent test cases, resulting in a speedup of up to 21.8× in test time compared to sequential testing. However, the fine-grained testing requires a large number of SUs, leading to a relatively large area overhead and a longer critical path.

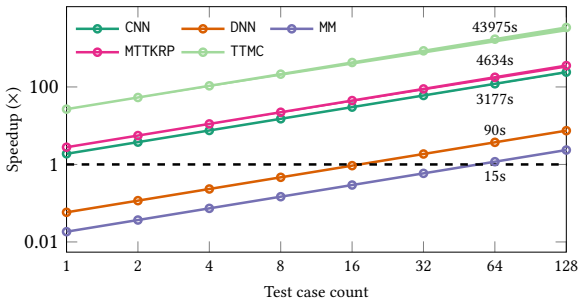


Figure 12: Speedup of PEDIA (including synthesis time) compared to Vivado XSIM. The numbers represent the average simulation latency per test case in XSIM.

Table 3: Average latency breakdown for each benchmark. CF = Configuration time of SUs. PF = Data prefetching time.

Benchmarks	# SU	Clock Cycles		Latency in %	
		CF	PF	CF	PF
CNN	3	2	10584	0.00	15.55
DNN	3	4	2048	0.00	1.53
MM	3	2	1296	0.02	12.89
MTTKRP	4	2	960	0.00	0.91
TTMC	4	2	8320	0.00	5.90
histogram	87	50	1001	0.30	6.09
fir	69	0	1000	0.00	6.37
iir	104	300	1001	1.62	5.41
image_resize	130	300	901	1.73	5.19
insertion_sort	204	160	1001	1.91	11.96
jacobi_1d	144	0	100	0.00	1.32
video_filter	209	320	1802	1.72	9.66

4.4 Testing Speedup

We now evaluate our approach in testing time compared with commercial HLS tools. Commercial HLS tools, such as Vitis HLS [1] and Stratus HLS [3], generate HLS designs in RTL code and orchestrate existing hardware simulators, such as Vivado XSIM and ModelSim, for testing. These simulators run test cases sequentially on a CPU, and PEDIA runs in parallel on an FPGA. FPGAs can achieve high performance but require synthesis time to generate the bitstream. For a fair comparison, we consider the total testing time, including both synthesis and run-time. The synthesis time for each benchmark is shown in Table 2. The proposed testing platform supports lightweight reconfiguration as described in Section 3.1, only requiring the generation of a single bitstream per benchmark. Detailed latency breakdown is shown in Table 3.

The test times per test case across different approaches are illustrated in Figure 10. Existing work on hardware-accelerated testing does not support HLS inputs, we use sequential testing as the baseline for hardware-accelerated testing and also compare PEDIA with two commercial HLS simulators, Vivado XSIM and ModelSim. In this figure, a lower bar indicates better performance. Overall, PEDIA achieves significant speedup compared to other approaches. The testing time of hardware-accelerated testing is faster than simulation on a CPU, as it exploits hardware parallelism and does not need to keep traces of signals at runtime. PEDIA achieves further speedup due to additional parallelism among test cases. There is

an interesting observation that PEDIA does not achieve hundreds of times speedup when testing hundreds of test cases in parallel. The main reason is that the delays introduced by SUs significantly extend the total latency of the HLS design, leading to a slowdown of up to tens of times. PEDIA hides such latency overhead by parallelism among test cases, still leading to significant speedup.

Despite the considerable time overhead caused by generating the bitstream, PEDIA still shows the fastest testing speed when running 1500 test cases, as shown in Figure 10. The speedup from PEDIA is expected to increase with more test cases. Figure 12 illustrates this trend in systolic array benchmarks, comparing the speedup by PEDIA on various test cases against Vivado XSIM. These benchmarks require long simulation time using Vivado XSIM due to the complexity of the HLS design. With a huge time offset of the synthesis time, PEDIA initially shows worse test times with fewer test cases on `MM` and `DNN`. However, as the number of test cases increases, the hardware testing latency hides the synthesis time, leading to a significant speedup on all benchmarks.

5 Related Work

HLS Programming Languages HLS tools have greatly improved hardware development productivity with software-defined hardware design. Various domain-specific languages (DSLs) have been introduced to bridge the gap between software programs and hardware designs. TAPA [14, 20] focuses on dataflow architectures and task-level parallelism optimization. Spatial [31] offers a data path-aware abstraction. PyLog [24] is a Python-based DSL for hardware mapping. HeteroCL [34] and HeteroFlow [50] express both hardware algorithms and optimizations. PolySA [16] and SODA [13] provide affine abstractions for hardware optimizations. However, these high-level abstractions lack cycle-sensitivity and rely on HLS tools for scheduling and re-timing. PEDIA presents a novel abstraction for HLS testing, offering an interface for cycle-sensitive analysis and testing of HLS designs.

HLS languages with hardware typing systems ensure hardware schedule correctness by construction. Dahlia [37] is a scheduling-aware DSL focusing on affine programs. Filament [38] uses timeline types to describe hardware computing IIs at the source level. Aetherling [17] introduces a strong type system for streaming architectures. However, all these typing systems cannot handle operations with data-dependent or variable latencies because of their limits in static analysis, while PEDIA offers a general approach for dataflow HLS designs including dynamically scheduled circuits.

HLS Testing and Verification Existing hardware testing primarily relies on RTL simulation [2, 4, 45, 49], which requires manual setup of testing environments. PEDIA addresses this by automatically generating latency insensitivity test cases for designers, simplifying HLS testing to resemble traditional software testing.

Most HLS testing frameworks rely on CPU-based hardware simulation. Kôika [12], a hardware DSL, utilizes Cuttlesim [41] for testing, transforming hardware designs into software specifications for debugging. Another hardware DSL, Chisel [9], employs HGDB [51] for source-level debugging. Commercial tools like Xilinx Vitis HLS [1], Stratus HLS [3], and Intel HLS compilers [26] use their own hardware simulators or commercial RTL simulator like ModelSim [4]. Efforts like FastSim [5], FLASH [15], and LightningSim [42] have been made to speed up HLS testing by fast simulation. They still focus on varying input values and test on CPUs, while PEDIA provides latency insensitivity testing on FPGAs.

Hardware-accelerated Testing FPGA-based acceleration for hardware testing, especially at the RTL level, has been widely studied. The first specialized architecture for emulation was proposed by IBM [10, 40]. Initial FPGA-based emulators [8, 30, 32] are used to emulate low-level hardware designs. Tiwari *et al.* [47] propose using additional 'scan chains' circuitry for mapping and debugging hardware designs on FPGAs. Commercial tools, such as Intel Signal Tap Logic Analyzer [27] and AMD Xilinx ChipScope Integrated Logic Analyzer [6], provide interfaces for monitoring hardware behaviors on FPGAs at run-time. Further advancements include combining hardware emulation with software simulation using JIT-like methods for faster emulation [7, 43], as well as efficient architectures like ASH by Elsabbagh *et al.* [19] for accelerating RTL simulation through hardware-software co-design. However, these methods focus on low-level hardware languages and require manual specification of test cases. PEDIA targets high-level software programs and automatically generates cycle-sensitive test cases.

ENCORE [44] and FireSim [29] are FPGA-based frameworks for hardware testing and simulation using high-level software specifications. However, they rely on user effort for optimizations while PEDIA automatically optimizes both the test space and the hardware testing platform for efficient testing.

6 Conclusions

This paper addresses a gap in HLS testing where existing HLS testbenches insufficiently test the latency insensitivity of dataflow HLS designs, risking potential design bugs. We introduce PEDIA, an end-to-end toolflow for testing dynamic external data behaviors around the HLS design. The main contributions of PEDIA include its expansion of the test space beyond input value variation to include input arrival times, improving the robustness and reliability of dataflow HLS designs. This is especially valuable in real-world scenarios with dynamic inputs. The proposed cycle-sensitive abstraction in PEDIA enables users to effectively test crucial hardware behaviors for special use cases, such as late inputs and backpressure, overcoming limitations of current HLS languages. PEDIA leverages FPGA-based parallel testing for hardware acceleration, greatly reducing testing times and outperforming existing commercial hardware simulators. The reported average speedup of 4.53× indicates the effectiveness and efficiency of the proposed testing platform. Our future work will explore the fundamental limits of this approach, both theoretically and practically. First, we plan to add support for HLS testing on multiple FPGAs, migrating PEDIA to FPGAs in the cloud for device-level parallelism. Second, we plan to study coverage analysis for automated HLS testing. We plan to build on previous coverage analysis work [33], lifting it from the RTL level to the HLS level.

Acknowledgments

We thank Yann Herklotz for his assistance in finding HLS bugs and improving the quality of the paper. This work was supported by the National Key Research and Development Program of China (Grant No.2023YFB4405105) and the Major Program of the National Natural Science Foundation of China (Grant No.62090023).

References

[1] 2024. AMD Xilinx Vitis HLS. https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html

[2] 2024. AMD Xilinx Vivado. https://www.xilinx.com/products/design-tools/vivado.html

[3] 2024. Cadence Stratus HLS. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[4] 2024. ModelSim HDL simulator. https://eda.sw.siemens.com/en-US/ic/modelsim/

[5] Mohammed Abderehman, Jayprakash Patidar, Jay Oza, Yom Nigam, TM Abdul Khader, and Chandan Karfa. 2022. FastSim: A Fast Simulation Framework for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 5 (2022), 1371–1385. doi:10.1109/TCAD.2021.3090339

[6] AMD. 2024. ChipScope Integrated Logic Analyzer (ILA). https://www.xilinx.com/products/intellectual-property/chipscope_ila.html

[7] Sameh Attia and Vaughn Betz. 2020. StateMover: Combining simulation and hardware execution for efficient FPGA debugging. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 175–185.

[8] Jonathan Babb, Russell Tessier, Matthew Dahl, Silvina Zimi Hanono, David M Hoki, and Anant Agarwal. 1997. Logic emulation with virtual wires. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16, 6 (1997), 609–626.

[9] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. 1216–1225.

[10] Daniel K Beece, G Deiberg, Georgina Papp, and Frank Villante. 1988. The IBM engineering verification engine. In *25th ACM/IEEE, Design Automation Conference. Proceedings 1988.* IEEE, 218–224.

[11] Benjamin Biggs, Christos-Savvas Bouganis, and George Constantinides. 2023. ATHEENA: A Toolflow for Hardware Early-Exit Network Automation. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 121–132.

[12] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The essence of Bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 243–257.

[13] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[14] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. 2021. Extending high-level synthesis for task-parallel programs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 204–213.

[15] Young-Kyu Choi, Yuze Chi, Jie Wang, and Jason Cong. 2020. FLASH: Fast, Parallel, and Accurate Simulator for HLS. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 4828–4841. doi:10.1109/TCAD.2020.2970597

[16] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[17] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 408–422.

[18] Ayatallah Elakhras, Andrea Guerrieri, Lana Josipovic, and Paolo Ienne. 2024. Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 44–54.

[19] Fares Elsabbagh, Shabnam Sheikhha, Victor A Ying, Quan M Nguyen, Joel S Emer, and Daniel Sanchez. 2023. Accelerating RTL Simulation with Hardware-Software Co-Design. In *Symposium on Microarchitecture (MICRO'23)*.

[20] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, et al. 2023. TAPA: a scalable task-parallel dataflow programming framework for modern FPGAs with co-optimization of HLS and physical design. *ACM Transactions on Reconfigurable Technology and Systems* 16, 4 (2023), 1–31.

[21] NB Harshitha, YG Praveen Kumar, and MZ Kurian. 2021. An Introduction to Universal Verification Methodology for the digital design of Integrated circuits (IC's): A Review. In *2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS)*. IEEE, 1710–1713.

[22] Yann Herklotz, James D Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal verification of high-level synthesis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.

[23] Alex Horn, Michael Tautschnig, Celina Val, Lihao Liang, Tom Melham, Jim Grundy, and Daniel Kroening. 2013. Formal co-validation of low-level hardware/-software interfaces. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 121–128.

[24] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen-mei Hwu. 2021. Pylog: An algorithm-centric python-based FPGA programming and synthesis flow. *IEEE Trans. Comput.* 70, 12 (2021), 2015–2028.

[25] Alaa Hussien, Samar Mohamed, Mohamed Soliman, Hager Mostafa, Khaled Salah, Mohamed Dessouky, and Hassan Mostafa. 2019. Development of a generic and a reconfigurable UVM-Based verification environment for SoC buses. In *2019 31st International Conference on Microelectronics (ICM)*. IEEE, 195–198.

[26] Intel. 2024. High-Level Synthesis Compiler. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[27] Intel. 2024. Signal Tap Logic Analyzer: Introduction & Getting Started. https://www.intel.com/content/www/us/en/docs/programmable/683819/23-3/faq.html

[28] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 127–136.

[29] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. 2018. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 29–42.

[30] Ubaid R Khan, Henry L Owen, and Joseph LA Hughes. 1993. FPGA architectures for ASIC hardware emulators. In *Sixth Annual IEEE International ASIC Conference and Exhibit*. IEEE, 336–340.

[31] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 296–311.

[32] Helena Krupnova and Gabriele Saucier. 2000. FPGA-based emulation: Industrial and custom prototyping solutions. In *International Workshop on Field Programmable Logic and Applications*. Springer, 68–77.

[33] Kevin Laeufer, Vighnesh Iyer, David Biancolin, Jonathan Bachrach, Borivoje Nikolić, and Koushik Sen. 2023. Simulator Independent Coverage for RTL Hardware Languages. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 606–615.

[34] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 242–251.

[35] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5. 1–20.

[36] Matthew Naylor and Simon Moore. 2015. A generic synthesisable test bench. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. 128–137. doi:10.1109/MEMCOD.2015.7340479

[37] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 393–407.

[38] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular hardware design with timeline types. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 343–367.

[39] Non-Blocking API in Vitis HLS. 2024. https://docs.amd.com/r/en-US/ug1399-vitis-hls/Non-Blocking-API

[40] Gregory F Pfister. 1982. The yorktown simulation engine: Introduction. In *19th Design Automation Conference*. IEEE, 51–54.

[41] Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, Arvind, and Adam Chlipala. 2021. Effective simulation and debugging for a high-level hardware language using software compilers. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 789–803.

[42] Rishov Sarkar and Cong Hao. 2023. LightningSim: Fast and Accurate Trace-Based Simulation for High-Level Synthesis. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 1–11. doi:10.1109/FCCM57271.2023.00010

[43] Eric Schkufza, Michael Wei, and Christopher J Rossbach. 2019. Just-in-time compilation for Verilog: A new technique for improving the FPGA programming experience. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 271–286.

[44] Kan Shi, Shuoxiang Xu, Yuhan Diao, David Boland, and Yungang Bao. 2023. ENCORE: Efficient Architecture Verification Framework with FPGA Acceleration. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 209–219.

[45] Wilson Snyder. 2004. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*.

[46] Synopsys HECTOR. 2024. https://www.synopsys.com/verification/static-and-formal-verification/vc-formal/vc-formal-datapath-validation.html

[47] Anurag Tiwari and Karen A Tomko. 2003. Scan-chain based watch-points for efficient run-time debugging and verification of FPGA designs. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*. 705–711.

[48] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 93–104.

[49] Stephen Williams and Michael Baxter. 2002. Icarus verilog: open-source verilog more than a year later. *Linux Journal* 2002, 99 (2002), 3.

[50] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. 2022. HeteroFlow: An accelerator programming model with decoupled data placement for software-defined FPGAs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 78–88.

[51] Keyi Zhang, Zain Asgar, and Mark Horowitz. 2022. Bringing source-level debugging frameworks to hardware generators. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1171–1176.