HLS_PRINT: High Performance Logging Framework on FPGA

Nupur Sumeet
Tata Consultancy Services - Research
Mumbai, Maharashtra, India
nupur.sumeet@tcs.com

Manoj Nambiar
Tata Consultancy Services - Research
Mumbai, Maharashtra, India
m.nambiar@tcs.com

ABSTRACT

FPGAs have been tipped to be useful for implementing low latency transaction processing systems. Getting computationally powerful over time, they are making their way into enterprise data centers. Another factor is the availability of C compilers for FPGAs as opposed to hardware description languages (HDLs) that requires special skills. However, data center operations staff were concerned about real time troubleshooting in production. Tracing FPGA implemented application execution require special skills and vendor specific tools that can capture limited by the amount of data.

To address this, we designed and implemented a logging framework on the FPGA. This paper presents the design and implementation of the framework. We present an algorithm that checks and generates alerts for performance overheads introduced due to the use of logging. Finally, experimental results are presented which demonstrate zero or low overhead of the logging framework.

CCS CONCEPTS

• **General and reference → Evaluation**; • **Hardware → Board- and system-level test**.

KEYWORDS

FPGA logging, High performance logging, HLS log framework

ACM Reference Format:
Nupur Sumeet and Manoj Nambiar. 2021. HLS_PRINT: High Performance Logging Framework on FPGA. In *Proceedings of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3427921.3450238

1 INTRODUCTION

Recent availability of High-Level Synthesis (HLS) development flow from FPGA vendors like Xilinx [18] and Intel [7] have simplified hardware design development to a great extent. A HLS design development flow includes a compiler which can compile a high-level language, such as C/C++, into the corresponding HDL (Hardware Description Language) representation. However, this includes limited support for the standard libraries that accompany these languages. Nevertheless, one can expect to write application functionality in C and have it implemented on an FPGA without having to know RTL (Register Transfer Logic) design or HDL.

Developing applications using HLS would entice enterprise users, given the simplicity of coding in a high-level language. However, there was a practical problem that came up. The data center operations staff were very well used to monitoring software applications in the data center. Almost all applications running in the data center would log important data. This included the run time contextual data, intermediate steps and final results - depending on the level of logging in force. This information could serve many purposes like auditing, data for machine learning or just troubleshooting application execution issues. The last requirement is very essential to ensure reduced downtime as availability issues could result in significant loss in business. But this was not possible with FPGA application developed even in HLS. Collecting data which involved the run time execution context of applications on FPGA required use of hardware vendor specific modules called integrated logic analyzers (ILA)[16]. The following issues are associated with use of ILAs:

1. Manual integration in the HDL representation that is tedious and calls for automation.
2. The use of ILAs required connection with the FPGA's Integrated Development Environment (IDE). This requires knowledge of HDL and the advantage that HLS enables is lost.
3. The ILAs have an associated buffer which has an upper limit on amount of data that can be captured. This limit is too less for collecting information about events in production.

Addressing these issues, we built a logging framework which would enable logging for FPGA implemented applications just as they would for software-based applications. It would be simple to use and the developers could have control on what data to log in the C/C++ (HLS) code. The data would be available to operations staff in similar form as software applications with regard to limitations on the amount of data to be logged. The logging framework is very similar to the use of printf function available in the C *stdio* library that comes with standard C-based software compilers.

Given the parallelism available in the FPGAs, it was desirable that the logging takes place in parallel with the hardware design execution, thereby causing zero performance overhead to the application. This is not the case for logging in software where there is always some performance overhead associated. To this effect, we have implemented an algorithm which checks for the loss of performance due to logging and alerts the user in such a case. Also included is the sizing of queuing resources used for logging to minimize performance impact.

ICPE '21, April 19–23, 2021, Virtual Event, France

ACM ISBN 978-1-4503-8194-9/21/04…$15.00
https://doi.org/10.1145/3427921.3450238

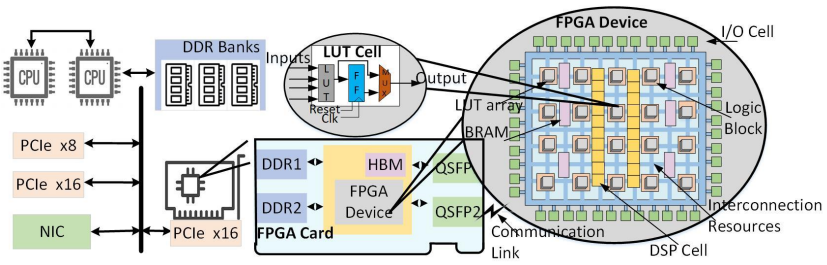

Figure 1: FPGA Resource and Datacenter Ecosystem

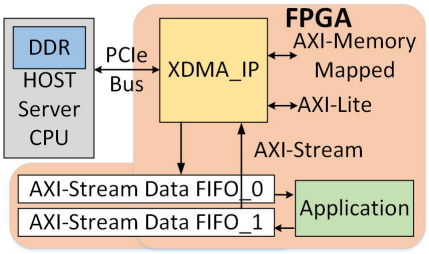

Figure 2: The Application (loop-back through AXI-Stream Data FIFO) communicating with the Host through XDMA_IP

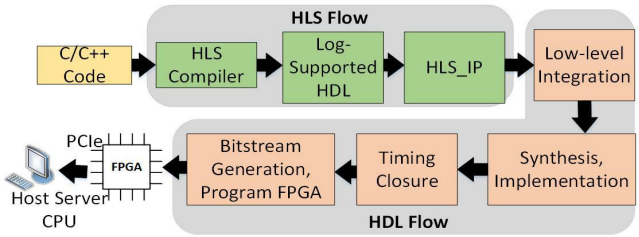

Figure 3: The Traditional HLS-HDL flow diagram.

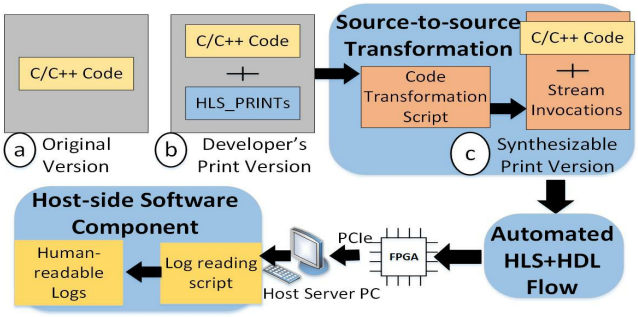

Figure 4: The conceptual flow diagram of HLS_PRINT framework.

In this paper, we propose a hardware-software solution to enable print functionality in the HLS design platforms to bridge the hardware logging gap. The proposed framework, named as HLS_PRINT enables print support and offers the possibility of zero latency overhead on application. We use source-to-source transformations such that the print statements in the source code results in HLS synthesizable constructs. In addition to this, the HLS_PRINT offers a push-button integration into an existing HDL project. The software part of the framework receives data from FPGA and presents it in human readable format.

The Key Contributions of this work are the following:

1. Development of a Logging Framework for HLS based FPGA application with the following benefits
   a. Completely automated –avoid developer's engagement in implementation complexities
   b. Software-like Logging –enables software experienced developers to develop and troubleshoot on FPGA platforms
   c. Alerts user for Performance overheads introduced by printing using HLS_PRINT
2. Recommended configuration of the Framework to avoid performance overheads while ensuring lossless printing.
3. Evaluation and Analysis of the HLS_PRINT on an industrial transactional application and open-source MachSuite Benchmarks.

The rest of the paper is organized as follows. The HLS_PRINT framework is discussed in Section 3. In Section 4, MachSuite benchmarks [14] and an industrial transactional application are used to evaluate the proposed framework followed by related work in Section 5. The Limitations of framework and Conclusion are discussed in Sections 6 and 7, respectively.

2 BACKGROUND: FPGA CONCEPTS

In this section, we introduce few core FPGA concepts in brief. These concepts are referred to at various places in the paper.

2.1 FPGA Resources and Data Center Ecosystem

FPGAs have entered the datacenter space with CPUs and are increasingly supporting complex algorithms on its own or through hybrid systems. The representative diagram of a data center FPGA is shown in Figure 1. The FPGA device, memories (on-chip and off-chip), network ports *etc.* form an FPGA sub-system that interacts with CPU and peripherals. The FPGA device is the programmable silicon that realizes the desired functionality. This entails a matrix of logic blocks (Look-up table (LUT) arrays, Block RAMs, Digital Signal Processor (DSP), Flip-flops (FFs), Multiplexers) connected through programmable interconnects and I/Os.

2.2 Traditional HLS+HDL Design Flow

Figure 3 shows the HLS+HDL design flow. The HLS compiler converts application source code developed in high-level language to low-level HDL implementation. The implemented HDL is packaged

and exported as an Intellectual Property (IP) for use with HDL design flow tools. The HDL design flow includes the low level integration of HLS_IPs with vendor-specific IPs to create an end-to-end working design. Following the low-level integration, the design is synthesized and implemented while meeting the desired frequency for performance. The FPGA is then programmed with the generated bitstream.

2.3 DMA Transfer: XDMA_IP as PCIe endpoint

The Xilinx DMA (XDMA_IP) Subsystem for PCI Express implements a high-performance DMA for use with the PCIe (Peripheral Component Interface Express) Block. The XDMA_IP allows for the bi-directional movement of data between Host memory and the DMA (Direct Memory Access) subsystem. These DMA engines can be mapped to individual Advanced eXtensible Interface (AXI)-Stream interfaces [1] or a shared AXI4 memory mapped (MM) interface to the user application.

2.4 AXI-Stream Data FIFO IP

The FIFO module is capable of providing temporary storage (a buffer), with depth limit 16 to 32768, for logged data before transmitting it to the CPU via the XDMA_IP. The input and output ports of the FIFO can be configured to operate at independent frequencies. This ensures that the application can be synthesized at a range of frequencies and is not limited by XDMA_IP operating frequency *i.e.* 250 *MHz*. Figure 2 shows XDMA_IP connected to a transactional application using AXI stream FIFOs.

3 THE PROPOSED HLS_PRINT FRAMEWORK

During synthesis, the HLS compiler ignores the print statements in the source code. In our approach, we use source-to-source transformations to pre-process the application code such that the HLS compiler generates a synthesizable construct for print statements. The conceptual flow diagram of the framework is shown in Figure 4. The source-to-source transformations are automated using a code generation script. HLS_PRINT can be easily ported to other HLS frameworks, commercial or otherwise as it works on at a pre-compile stage.

3.1 User View: Source-to-Source Transformation

HLS_PRINT takes the form of a new directive which contains the name (-var) and type (-typ) of the print variable. It also supports the field (-com) for string/sentence that the user wants to print along with the print data. This feature enables greater user readability and control over logging as the developer is able to draw a clear correspondence between the print data received and the position at which the print was inserted in the application code. A sample application code is shown in Figure 5 (a), (b) and (c) as original, developer's print and synthesized print versions, respectively. The developer adds the HLS_PRINT statements to the application code (developer's print version in Figure 5 (b)) and executes the code generation script. The output of this script is the synthesizable print version as shown in Figure 5 (c). The script identifies the HLS_PRINT statements and pushes the copy of the variable to be printed to the HLS stream variable. The variables (of type"stream" in Figure 5 (c)) are also defined as arguments in the function. This enables the HLS compiler to expose the stream variables as interfaces that can be connected to other modules of HLS_PRINT. In the example shown in Figure 5, we are able to capture the changes in the variable *a* and *c* as the code executes.

3.2 Types of variables supported for printing

The print framework supports a range of datatypes - char, short, integer, double, float, fixed and arbitrary precision datatypes. In case of fixed or arbitrary precision datatype variables, the value of N is limited to 55bytes/stream as 9 bytes are kept for reserved for print data annotations discussed in subsection 3.4.1. The structure containing these data types are also supported. HLS_PRINT supports loop printing and the print data is pushed into the stream *n* times where *n* denotes the loop count.

3.3 Methodology for supporting Print in HLS compilers

Similar to software design, the application in hardware design is generally broken into smaller pieces called modules where each module is designated with a sub-task. The modules communicate with each other through input/output ports and only the signals forced on these ports are accessible for data/control transfers between modules. Following this concept, the application data that needs to be printed must be forced on the output port of the module. Using HLS pragma, the output port is further defined as AXI stream interface [1] as it supports bulk transfer of data from FPGA to Host/PC using the DMA (direct memory access) transfer. The HLS compiler implements stream variables as First-In-First-Out (FIFO) queue that permits one write/read per clock cycle. The RTL diagram of print supported code, print data FIFOs for variables *a* and *c* and the clock-wise operation scheduling is shown in Figure 6. The HLS compiler schedules operations to extracts maximum parallelism. The variable value update and write to stream variable operations are scheduled in the same clock cycle by the compiler as it sees them as independent operations. We leverage this compiler behavior for HLS_PRINT and thus, incur no additional latency for printing.

3.4 HLS_PRINT: Hardware Architecture

In general, an ideal architecture for logging framework would ensure that all print records of all variables are printed (lossless printing) with no additional latency to the applications run time. The HLS_PRINT architecture is developed considering these primary requirements.

Figure 7 is a representative hardware architecture of HLS_PRINT framework. The HLS IP blocks represents C/C++ functions synthesized with HLS compiler. Every variable printed by the function has a dedicated FIFO. Because of separate interfaces for every print variable, multiple stream variables can be updated without contention and no latency overhead is encountered. The FIFOs are arbitrated by the print multiplexing unit (PMU) (discussed in subsection 3.4.2) for delivering logged data to host via the XDMA_IP. The FIFOs receiving the print data can become full when its writing rate exceeds the reading rate. To ensure lossless print in such a situation, the

```
1
2  void func_foo() {
3      int A=5,B=6,C=7,i=0;
4      printf("Val A is %d",A);
5      A=B+C;
6      printf("Change1:A %d",A);
7      A=2*A;
8      printf("Change2:A %d",A);
9      while(i<10) {
10         C=C++; i++;
11         printf("Val C %d",C);
12     }
13 }
```

```
void func_foo() {
    int A=5,B=6,C=7,i=0;
    HLS_PRINT -var A -typ int -com Val A
    A=B+C;
    HLS_PRINT -var A -typ int -com Change1:A
    A=2*A;
    HLS_PRINT -var A -typ int -com Change2:A
    while(i<10) {
        C=C++; i++;
        HLS_PRINT -var C -typ int -com Val C
    }
}
```

```
void func_foo(stream <int> prt_stm_A,
     stream <int> prt_stm_C) {
    int A=5,B=6,C=7,i=0;
    prt_stm_A.write(A);
    A=B+C;
    prt_stm_A.write(A);
    A=2*A;
    prt_stm_A.write(A);
    while(i<10) {
        C=C++; i++;
        prt_stm_C.write(C);
    }
}
```

Figure 5: Sample Application code in (a) Original (b) Developer's Print and,(c) Synthesizable Print versions.

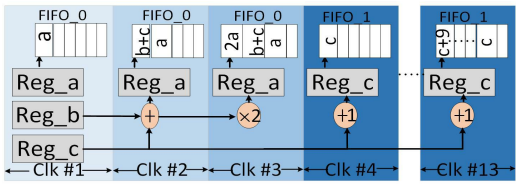

Figure 6: RTL diagram for Print Supported Code.

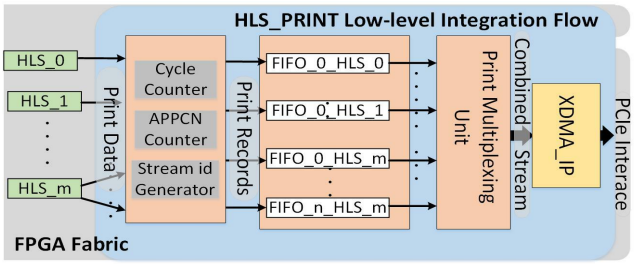

Figure 7: HLS_PRINT: Multiple Stream Configuration.

application has to stall till there is space in the FIFO to continue accepting print data. We discuss a scheme for FIFO sizing based on its access rates in subsection 3.6 to ensure lossless print while avoiding application stalls.

3.4.1 Print Record Format. Before writing the FIFOs, the print data is annotated with <*stream_id, APPCN, clock_cycle, print_data*>. The annotated print data is denoted as print record. A unique stream_id is assigned to every variable for correlating the HLS_PRINT directive with print data in the receiver software. The function invocation (or call) number (APPCN) is implemented as a hardware counter which is incremented once for every function call. The clock cycle counter counts clock cycles starting from system reset. The value of this counter is recorded every time data is pushed into the FIFO and denotes the clock_cycle. Stream_id, APPCN and clock_cycle are populated by HLS_PRINT using handshaking signals available in HLS IP. The XDMA_IP takes 64 bytes print record in one clock cycle out of which print data annotations takes 9 bytes and remaining 55 bytes are reserved for print data coming from the HLS_IP. The cycle counter and APPCN counter are 32-bit wide. In case multiple HLS functions contain HLS_PRINT directive, an additional *function_id* can be appended to print record for print-function association.

3.4.2 Print Multiplexing Unit (PMU): Stream Combining Scheme. The *n* FIFO queues are combined using Print Multiplexing Unit (PMU). The purpose of multiplexing unit is to scan FIFOs for valid data and forward it XDMA_IP for transfer. This is done based on APPCN. The prints belonging to the same APPCN are combined and transferred from FPGA. This helps ensure that the print record passed to host over PCIe does not overlap function call invocations. By doing this, we reduce the sorting complexities at the host end, as the sorting needs to be done only within the APPCN. The *n* variable FIFOs are scanned for valid data in round robin manner, based on

Algorithm 1 Print Multiplexing Unit

For a HLS function, n = # variable FIFOs, APPCNT = current function invocation count, $FIFO_j$ = FIFO for print variable #j
Initialize record with print record and APPCNT to 1
while forever **do**
  for j in n **do**
    while record in $FIFO_j$ **do**
      if record.APPCN == APPCNT **then**
        Dequeue record; Enqueue to XDMA_IP
      else
        Break
      end if
    end while
    APPCNT++
  end for
end while

Algorithm 1. For simplicity, we assumed only one HLS function with HLS_PRINT directive is present.

3.5 Host-side Software

The FPGA starts its computations when the bitstream is loaded into it and data can be transferred over the PCIe interface using the a Xilinx DMA driver.

Print Receiver Software Component: The receiver software receives the print data from the FPGA over PCIe in non blocking mode and saves the data in binary files as they arrive.

Formatter Process: The second software component asynchronously reads the binary files and processes them as shown in Algorithm 2.

The print record annotations are used for sorting based on APPCN, cycle counter and stream_id. Each queue has a unique stream_id and hence can be correlated to its corresponding HLS_PRINT directive. The -com field (shown in Figure 5) string can be printed beside the print data gives a software-like print experience. Note that the use of same FIFO for pushing updated values of *a* in Figures 6 and 5(c) is only for compact representation and the actual implementation contains dedicated FIFO for every HLS_PRINT directive. Here again, we assumed only one HLS function with HLS_PRINT directive is present.

Algorithm 2 Formatter Process

```
Initialize APPCNT to 0 and record_set to NULL
record = read first print record from file
while forever do
  APPCNT = record.APPCN
  while APPCNT == record.APPCN do
    Add(record_set, record)
    record = read next print record from file
  end while
  Sorted_records = Sort_on_cycle_count_field(record_set)
  for k in sorted_records do
    print k.clock_cycle, k.stream_id.comment, k.print_data
    // k.stream_id.comment indicates -com field
  end for
  record_set = record // initialize record_set
end while
```

3.6 Variable print stream FIFO sizing

While ensuring that there is no loss of data to be logged, it is also desirable that the inserts into the variable FIFO do not stall the application execution. For this, it is essential to right size the maximum depth of the variable FIFOs. Given the PMU functionality, we approximated each variable FIFO as an M/M/1 queue [5]. At the servicing end of the queue, XDMA_IP channel interface can consume 1 print record of a printed variable (64 bytes regardless of the size of the variable) every clock cycle. The XDMA_IP consume 8 print records at a stretch and introduces one unused cycles after the transfer due to the XDMA_IP constraints [17]. So, the total service rate for serving all the variable FIFOs combined is μ= (8/9 × clock_period). In this case, clock_period is 4 *n*s owing to the XDMA_IP which is synthesized to operate at 250 *MHz*. For a print variable v_i, c_i denotes the print count of v_i. Let's also denote the total number of print records in HLS function as $y = \Sigma(c_i)$. Hence, for a single variable FIFO, only a fraction of the service rate will be applicable which can be represented as a weight w_i of the variable. This is calculated as $w_i = c_i/(y)$. Therefore, the service rate available for the variable FIFO is estimated as $\mu_i = \mu$ x w_i = (8 x w_i) / (9 x clock_period).

The arrival rate of the variable FIFO can be calculated as highest frequency at which HLS function invokes the printing v_i, which we denote as λ_i. If the minimum latency of the function is z clocks, then $\lambda_i = (c_i)/(z$ x clock_period). Please note that while the XDMA_IP can operate at a fixed frequency 250 *MHz*, the HLS module can be synthesized to operate at any frequency *F MHz*. For sizing the variable FIFO, we need to express the arrival rate as observed by the XDMA_IP. So, this latency is scaled by 250/*F*. So, the effective latency zeff = z x 250/*F*. Therefore, the arrival rate $\lambda_i = c_i/$ (zeff x clock_period).

Lets define $\rho_i = \lambda_i/\mu_i$. For HLS_PRINT operation without interrupting the application, we require ρi < 1. This is the maximum rate at which the function can be invoked, but we will use this as the average arrival rate for use with M/M/1. For an M/M/1 queuing system the mean number in the system is defined as $\rho_i/(1\text{-}\rho_i)$. This is a conservative measure as it can be seen that λ_i is not really an average value, but a maximum value. For our variable FIFO ρ_i = (9 x y)/ (8 x zeff). We can see that this expression is the same for all variables as the service rate is proportional to the arrival rate. All variable FIFO can be set with max depth of (9 x y)/((8 x zeff) - (9 x y)) print records. Let's call this equation (1). Note that this is dependent only on the total number of print records and the effective latency of HLS function in context.

Revisiting Figure 5 (c), the variables *a* and *c* are stored on FIFO_0 and FIFO_1 (see Figure 6). c_a is 3 and c_c 10 with the assumed operating frequency of 200 *MHz*. We compute the arrival rates, λa (=46.15 records/μsec) and λc (=153.85 records/μsec). The total arrival rate is 200 records/μsec is less than service rate μi is 222.2 records/μsec. Using equation (1) with z = 13 clocks, the optimal queue size is 9-deep for both FIFOs.

3.7 HLS_PRINT Automation Algorithm

As can be seen from Figure 3, printing variables in HLS code in an existing FPGA project is very tedious when done manually. To address this, we have built an automation script that takes the following inputs (1) HLS_PRINT enabled Project folder Path (FPATH), (2) Min frequency of operation (MINFREQ), (3) Expected (average) call Rate (EXPCR), and (4) Expected # print variables in HLS function (EXPPRNT). We assume that the original project (without the HLS_PRINT directives) has been successfully synthesized.

The flow chart of the automation script is shown in Figure 8. The transformations step replace the HLS_PRINT directives with relevant code as shown in Figure 5 (c). The application print bandwidth (APPBW) is calculated as EXPCR × EXPPRNT. In case APPBW exceeds the system PCIe transfer bandwidth, user is alerted of a possible performance slowdown due to print support (pass decision box after "compute Print BW" branching to 'no' in Figure 8). Low-level integration includes integration of HLS IP with FIFOs, PMU and XDMA_IP (as in Figure 7) and setting pin constraints. The algorithm assumes that all the variables being printed are in single HLS module, in the project, however it can be extended to include variables being printed from multiple modules HLS or otherwise with additional module relevant inputs from the developer.

4 TESTS, RESULTS AND ANALYSIS

We evaluate the HLS_PRINT framework on MachSuite Benchmarks [14] provided for accelerator design in HLS design environment. We used the Vivado HLS 2019.2 for HLS and Vivado Design Suite 2019.2 for HDL design flow on Xilinx Alveo U280 FPGA board. The HLS design flow offers C-simulation and co-simulation for design analysis. Co-simulation provides hardware signal plot in the form is waveforms, which is a typical way of analyzing hardware design

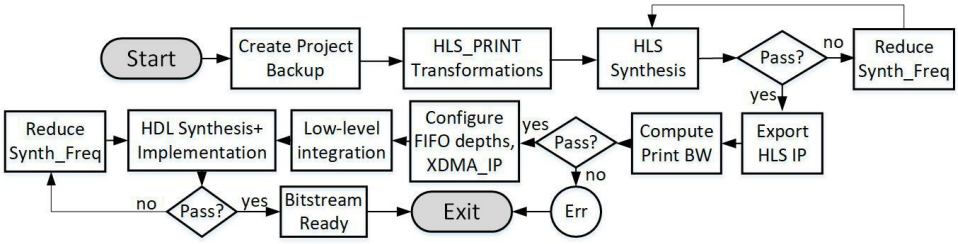

Figure 8: HLS_PRINT automation Algorithm flow chart.

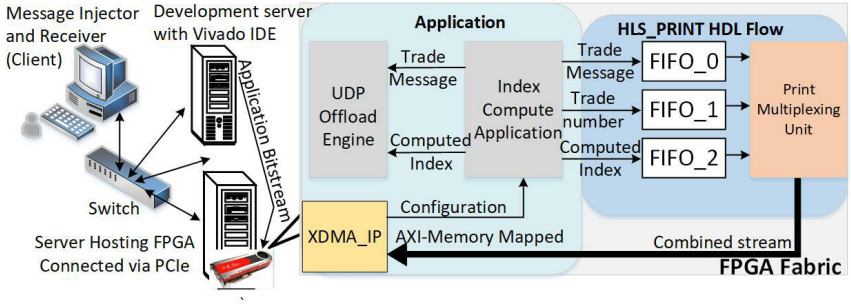

Table 1: HLS_PRINT Resource Overhead

	LUT	FF	BRAM
XDMA	65259	61532	124
FIFO_0/1 (104-bitx16-deep)	159	412	0
FIFO_2 (152-bitx16-deep)	191	532	0
Index compute*	17197	24651	5
Logic Analyzer@	3928	7762	41

∗ marked is Application

Figure 9: System-level test setup for Index Compute operation on FPGA.

execution. We evaluate the effect of HLS_PRINT on benchmark kernel functionality and latency. We performed, the hardware testing for industrial application and GEMM from MachSuite to further corroborate our HLS results when the application is running in real-time. For the purpose of performance testing, we added these clock cycle counters as counters in the low-level HDL of application under test. The counters are used to calculate the throughput and latency of the functions on the FPGA. ILAs [16] are used to observe the counters.

4.1 Index Compute Application

The stock exchange index compute is an industrial application that operates in client-server based architecture as shown in Figure 9. The expected response time of these applications is in μsec range. Due to the increasing volume of requests and fast response requirements, a paradigm shift from software to hardware based solution is observed. The IC algorithm details are proprietary nature and hence can not be discussed in the paper. Instead we discuss the index compute (IC) flow that consists typically of four stages. The trade message is received and processed by extracting trade parameters. The index compute algorithm is executed on the FPGA and the updated index is sent back. We ported the index compute algorithm to the FPGA using HLS to function as a server application. The trade messages are synthetically generated by the client system and are received by UDP offload engine [15] in the FPGA. The IC block contains configurable weights and an index seed value. The incoming trade messages contain various information such as token number, trade volume, trade price, type of trade *etc.* which identifies the traded stock. For the purpose of book-keeping, it may be necessary to log the input packets, the computed index and the number of trades processed. The logged data can be analyzed at the end of the trade-day or can be used to analyze the daily trade-traffic on hourly basis. In this test, we print incoming trade message (structure, 48 bytes), the number of processed trades (int) and index value (float) in IC using HLS_PRINT, verify the functionality, check for any change in overall performance and analyze the additional resource usage on FPGA as a result compared to the implementation without HLS_PRINT. The IC IP is synthesized at 156.25 *MHz* in HLS and integrated with low-level IPs. The performance (latency) counter accounts for cycles between receiving trade message and sending computed index back to UDP offload engine. The latency of original and HLS_PRINT supported index compute remains unchanged at 518.4 *ns*.

4.1.1 HLS_PRINT Resource Overhead on Index Compute. The HLS _PRINT framework uses XDMA_IP and AXI stream data FIFOs for low-level integration of HLS exported IP. The XDMA_IP is used for initial configuration of the index compute module and is the major resource contributor. We present the resource overhead due to these IPs (indicated in boldface) in Table 1. For comparison, we configured an ILA of 20 probes with 1024 depth and average width 72. The LUT, FF and BRAM utilization is considerable for ILA based logging. Though in comparison, HLS_PRINT consumes most resources, it provides a much longer signal visibility and with a higher transfer width of 512 bits.

4.2 MachSuite Benchmarks [14]

The MachSuite benchmarks [14] are synthesizable standard kernels crafted to emulate varied practical applications based on custom architectures and accelerator designs. MachSuite is a set of 19 benchmarks spanning 12 different kernels. Because of their diversity and

application coverage, we use the MachSuite benchmark kernels to evaluate the HLS_PRINT framework.

4.2.1 HLS_PRINT results on MachSuite. The MachSuite kernels are imported in Vivado HLS and synthesized for MOFREQ *MHz* frequency. For simplifying the hardware testing, input data array is hard coded into application code by minor changes in the code. We choose various datatypes for print variables ranging from small bit-width *e.g.* char (size-1 byte) to ap_uint<192> (size-192 bits). The highest number of print variables (262k) and print volume of 4.5MB per function call was configured in GEMM blocked kernel. The lowest (2685 cycles) and highest (67M cycles) latencies were seen for encryption and back-propagation benchmarks, respectively. Since there is no pipelining in the benchmarks, the latency extremes also indicates the throughput extreme cases. We test HLS_PRINT on all benchmarks by HLS synthesis and co-simulation. From the tests, we observe that the kernel functionality remains intact after adding HLS_PRINT statements as the HLS compiler resolves dependencies for print stream variable in the same way as it would do for any other variable. We observe that HLS_PRINT does not cause any reduction in MOFREQ with which the function was originally synthesized. The latency (in terms of clock cycle, LTCC@MOFREQ) of synthesizable print versions remained unaltered when MOFREQ timing constraint was enforced. Additionally, we observe that the required print data bandwidth does not exceed the system supported PCIe bandwidth for any kernel.

Among all MachSuite benchmarks, GEMM ncubedwas implemented on hardware for 1k function calls. We observe that there was no performance overhead and hardware results match with co-simulation. HLS_PRINT statements do not affect the latency of the GEMM kernel in real time measurements and the DMA transfer of print data is also successful.

The FIFO queues required for logging in MachSuite benchmarks are provisioned based on the optimal sizing guidelines discussed in section 3.6. The high latency and serviceable throughput (< 16 GB/sec) ensures use of minimum sized (16-deep) FIFO queues for MachSuite benchmark print variables. At this depth, the FIFO's utilize only LUTs and FFs. The maximum LUT and FF overheads are observed for MD_Grid kernel as 812 and 271, respectively. This amounts to negligible utilization of the available FPGA resources. Based on the conducted tests, it is observed that the proposed HLS_PRINT framework can support zero latency overhead prints for a wide variety of application kernels while retaining the kernel functionality. The stencil 3D kernel had the highest MOFREQ (1524 *MHz* and highest print bandwidth requirement of 4.5Gbits/s. Among all cases, we observe that tested HLS_PRINT configuration does not affect the latency or throughput of the original application. The results reinforces the confidence in use and configuration HLS_PRINT to support lossless logging with zero performance and minimal resource overhead.

5 RELATED WORK

We discuss related work based on two categories - 1. methods to transfer data between FPGA and host 2. methods enabling print in HLS designs. The comparison of HLS_PRINT with the related work is shown in Table 2.

Table 2: Comparison with State-of-the-Methods for Monitoring FPGA Application variables.

	Tran	DE	LL Int	OL Mont	R.Logs
Logic Analyzers [16]	×	HDL	M	✓	×
Xillybus[19]	×	HDL, HLS	M	✓	×
RIFFA[8]	×	HDL, HLS	M	✓	×
Calagar *et. al*[2]	SC	HLS	×	×	×
Goeders *et. al*[6]	SC	HLS	×	×	×
Jamal *et. al*[9]	SC	HLS	×	×	×
Monson *et. al*[11]	S2S	HLS	×	×	×
HLS_PRINT	S2S	HDL, HLS	Auto	✓	✓

Tran - Transformation used; DE - Design Environment; LL Int - Low-level Integration support; OL Mont - Online Monitoring; R.Logs - Readable Logs; SC - Source-code; S2S - Source-to-source; M - Manual; Auto - Automated;

Various PCIe based commercial products (by vendors Xillybus [19], PLDA) are available for seamless data transfer between FPGA and server. The Xillybus platform [19] interacts with the design's FIFO interface and facilitates data transfer. Authors in [8] presents an open source integration framework (RIFFA) for high performance FPGA accelerators. However, these solutions do not extend logging in the HLS environment. Their use in data transfer requires familiarity with complex low-level implementation details and time-consuming manual effort. HLS_PRINT provide automatic integration of print signals with low-level IPs. This makes the data transfer scheme fast and reduces human errors. The received data is annotated and presented in a human-readable format for a hassle-free software-like print experience. Additionally, none of these papers discuss impact on performance effect of these data transfer frameworks on high-throughput applications.

We discuss recent literature on enabling debug in HLS designs. Though debugging and logging are not equivalent, but design signals can be monitored (possibly for limited time-span) using debug tracing techniques. Recent works on extending debug feature in HLS compilers use source-level transformations [2, 6, 20] to gain visibility into design signals and monitors the mismatches between hardware and software executions. Authors in [6, 9, 10] use trace-buffer optimizations to capture signals for long time. These works use the intermediate representation (IR) of design to extract information required to introduce the debug circuitry in HDL. This limits their use to open-source HLS compilers only. None of the mentioned works, except for [9], extend debug support in production environments. Authors in [11], adopts source-to-source transformation to monitor debug signals. Their framework provides automation to bring signals embedded deep into HLS designs to the top module and mark them for debug. The user has to be familiar with abstract syntax tree representation of a program in order to use their framework. Additionally, work in [11] does not orchestrate the HLS-HDL integration, unlike HLS_PRINT which fully automates the task after the user adds HLS_PRINT directives in the code. The lack of a simple familiar easy to use interface like printf and full automation that will enable logging on FPGA without requiring developers and data center operations staff to have HDL skills motivated the development of HLS_PRINT.

Another interesting logging framework is SoCLog[13]. It generates activity logs that aids designers to identify performance bottlenecks and make efforts for load balancing, workload partitioning *etc.*

for HLS/HDL developed accelerators. The SoCLog scope is limited to performance logging whereas HLS_PRINT generates functional logs formatted by the developer and data center operations staff to record values of HLS variables implemented in the FPGA and can be easily used in data center production environments.

While we have discussed work related to logging in FPGA hardware implementations, there are well known logging frameworks in software. elog [12] supports c/c++ applications while log4j [3] is a popular logging framework for Java. The performance of log4j has been discussed in [4]. The 99%ile latency is reported to be 250 μs while the latency is negligible for most measurements. This can be expected in software logging implementations where there are system call and data copy overheads which always add to the application latency. On the FPGA with HLS_PRINT we have seen that it is possible to have zero latency logging overhead.

6 LIMITATIONS AND FUTURE WORK

We address the limitations of our work in this section with primary focus on the performance limiting compromises. We intend to address these limitations in future work

- *Frequency Compromise:* Case when original application (OA) is unable to sustain the synthesis (or implementation) frequency (SF) after print functionality (APIPF) in added. This could happen due to high resource utilization and high volume of print variables.
- *Latency Compromise:* The latency compromise indicates, the APIPF takes more clock cycles (CC) than that of the OA. HLS compiler will schedule 2 BRAM assignments in a clock cycle. Due to steam data type limitation, only one array assignment per clock cycle can be made. Additionally, HLS compiler will not schedule variable update and write to stream operations in one clock if they are far apart in the source code.
- *Throughput Limitations:* This limitation happens when the APIPF cannot be invoked at the same rate at the OA limited by the available system PCIe bandwidth.. Large number print variables and high print volume per function invocation can contribute to this.
- An unused DMA channel is required to implement the print functionality.
- Printing of stream variables (a data type available in HLS which represents a FIFO queue) is not supported at this time. The primary reason being that reading stream variable could cause it to be removed from the stream, thereby affecting application functionality.
- Inefficient use of PCIe bandwidth for pushing print data as discussed in subsection 3.6. It is possible to combine several print records into one, which is left for future work.
- Printing of arrays and elements larger than 55 bytes not yet supported.

7 CONCLUSION

In this paper, we discuss a Logging framework, HLS_PRINT, developed to support real-time logging in HLS applications ensuring minimum impact on application latency and throughput. The HLS_PRINT framework logging scheme is very similar to logging in software-applications. HLS_PRINT uses source code transformations and can operate with various HLS compiler without using compiler internals knowledge. The framework provides automation at various stages to ensure easy use and integration which can be easily used by a software engineer and operations staff at data centers without having hardware engineering skills. Moreover, the clock cycles information printed in the logs can be useful for performance analysis. The MachSuite benchmarks are used to evaluate the developed print framework where we study the effect on latency as well as the print traffic generated by the diverse kernels, under various printing configurations. We further report results on a transactional application and we can see from the hardware experiment results that it is possible to achieve zero latency impact prints making it a suitable candidate for logging in FPGA applications implemented using HLS.

REFERENCES

[1] ARM. 2010. AMBA 4 AXI4-Stream Protocol Version: 1.0. Retrieved June 15, 2020 from https://static.docs.arm.com/ihi0051/a/IHI0051A_amba4_axi4_stream_v1_0_protocol_spec.pdf

[2] N. Calagar, S. D. Brown, and J. H. Anderson. 2014. Source-level debugging for FPGA high-level synthesis. In Int. Conf. on Field Programmable Logic and Applications. 1–8.

[3] The Apache Software Foundation. 2020. Log4j. Retrieved December 15, 2020 from https://logging.apache.org/log4j/2.x/log4j-users-guide.pdf

[4] The Apache Software Foundation. 2020. Log4j. Retrieved December 15, 2020 from https://logging.apache.org/log4j/2.x/performance.html

[5] Natarajan Gautam. 2012. Analysis of Queues: Methods and Applications (1st ed.). CRC Press, Inc., USA.

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

[7] INTEL. 2019. INTEL® High Level Synthesis Compile. https://www.intel.in/content/www/in/en/software/programmable/quartus-prime/hls-compiler.html

[8] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. 2015. RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators. ACM Trans. Reconfigurable Technol. Syst. 8, 4, Article 22 (2015), 23 pages.

[9] Al-Shahna Jamal, Eli Cahill, Jeffrey Goeders, and Steven J. E. Wilton. 2020. Fast Turnaround HLS Debugging Using Dependency Analysis and Debug Overlays. ACM Trans. Reconfigurable Technol. Syst. 13, 1, Article 4 (2020), 26 pages.

[10] J. S. Monson and B. Hutchings. 2014. New approaches for in-system debug of behaviorally-synthesized FPGA circuits. In Int. Conf. on Field Programmable Logic and Applications. 1–6.

[11] J. S. Monson and B. Hutchings. 2015. Using source-to-source compilation to instrument circuits for debug with High Level Synthesis. In 2015 Int. Conf. on Field Programmable Technology. 48–55.

[12] Emanuele Oriani. 2020. elog. Retrieved December 15, 2020 from https://github.com/Emanem/elog

[13] Ioannis Parnassos, Panagiotis Skrimponis, Georgios Zindros, and Nikolaos Bellas. 2016. SoCLog: A real-time, automatically generated logging and profiling mechanism for FPGA-based Systems On Chip. 1–4.

[14] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In Intl. Sym.on Workload Characterization. 110–119.

[15] M. Ruiz, D. Sidler, G. Sutter, G. Alonso, and S. López-Buedo. 2019. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In Intl. Conf. on Field Programmable Logic and Applications. 286–292.

[16] Xilinx. 2014. Integrated Logic Analyzer v6.1 LogiCORE IP Product Guide. Retrieved July 10, 2020 from https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_1/pg172-ila.pdf

[17] Xilinx. 2016. UltraScale+ Devices Integrated Block for PCI Express v 1.3. Retrieved June 15, 2020 from https://www.xilinx.com/support/documentation/ip_documentation/pcie4_uscale_plus/v1_3/pg213-pcie4-ultrascale-plus.pdf

[18] Xilinx. 2019. Vivado Design Suite - HLx Editions. https://www.xilinx.com/products/design-tools/vivado.html

[19] Xillybus. 2020. Xillybus FPGA Designer's Guide. Retrieved June 5, 2020 from http://www.xillybus.com/downloads/doc/xillybus_fpga_api.pdf

[20] L. Yang, S. Gurumani, D. Chen, and K. Rupnow. 2016. AutoSLIDE: Automatic Source-Level Instrumentation and Debugging for HLS. In Int. Sym. on Field-Programmable Custom Computing Machines. 127–130.