HLS_Profiler: Non-Intrusive Profiling tool for HLS based Applications

Nupur Sumeet, Deeksha Deeksha, Manoj Nambiar
Computing Systems Lab, TCS Research
Mumbai, Maharashtra, India
{nupur.sumeet,deeksha.14,m.nambiar}@tcs.com

ABSTRACT

The High-Level Synthesis (HLS) tools aid in simplified and faster design development without familiarity with Hardware Description Language (HDL) and Register Transfer Logic (RTL) design flow. However, it is not straight forward to associate every line of source code to a clock-cycle of synthesized hardware design. On the other hand, the traditional RTL-based design development flow provides the fine-grained performance profile through waveforms. With the same level of visibility in HLS designs, the designers can identify the performance-bottlenecks and obtain the target performance by iteratively fine-tuning the source code. Although, the HLS development tools provide the low-level waveforms, interpreting them in terms of source code variables is a challenging and tedious task. Addressing this gap, we propose an automated profiler tool, HLS_Profiler, that provides performance profile of source code in a cycle-accurate manner. The HLS_Profiler tool is non-intrusive and collectively uses the ⟨static analysis, dynamic trace⟩ of the source code to present the performance profile report to attribute latent clock cycles to each line of source code. Additionally, we developed a set of associative rules to maintain correctness in performance profile of the HLS design. To verify correctness, we demonstrate the HLS_Profiler tool on MachSuite Benchmarks and an industry-grade recommendation application. The proposed HLS_Profiler framework provides visibility into the cycle-by-cycle hardware execution of source-code and aids the designer in making performance-centric decisions.

CCS CONCEPTS

• **Hardware → Software tools for EDA**; **Board- and system-level test**; • **Computing methodologies → Model verification and validation**; • **General and reference → Evaluation**.

KEYWORDS

Hardware profiling, HLS Designs, Performance Profile, Performance Analysis

ACM Reference Format:
Nupur Sumeet, Deeksha Deeksha, Manoj Nambiar. 2021. HLS_Profiler: Non-Intrusive Profiling tool for HLS based Applications. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22), April 9–13, 2021, Virtual Event, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3427921.3450238

ICPE '22, April 9–13, 2022, Virtual Event, China

ACM ISBN 978-1-4503-9143-6/22/04…$15.00
https://doi.org/10.1145/3427921.3450238

1 INTRODUCTION

Performance profilers are software development tools designed to assist in performance analysis of applications and improve poorly performing sections of code. They provide measurements on time-taken by a routine to execute, proportion of total time spent on it, its parent routine *etc.* These practices are quite common in software paradigm as matured profiling tools are available. However, it is not the case in hardware development, mainly FPGAs (Field Programmable Gate Arrays).

FPGAs are becoming a popular choice as an application accelerator due to its support for deep pipelines as well as low latency realisations. The datapath based design in FPGAs is favourable to performance sensitive applications. The traditional hardware development encompasses coding the design in Hardware Description Languages (HDLs), such as Verilog and VHDL, and define the Register Transfer Logic (RTL) datapath from input to output and achieve the desired functionality. A recent approach for hardware design development is through High Level Synthesis (HLS) tools [7, 17]. With HLS, the design development productivity improves as it supports high-level languages (C/C++) and the process of creating HDL description, defining RTL datapath, operation scheduling *etc.* are abstracted away from the developer. The HLS development flow includes HLS compiler for generating HDL description of source code and include co-simulation for cycle-accurate functional analysis as shown in Fig. 1. It should be noted that HLS development requires an additional step of implementation to generate the FPGA executable.

Although the HLS development flow provides a simplified, faster and highly abstracted way for hardware design developments, often, better parallel or pipelined algorithms may be designed which are better suited to the FPGA architecture [1]. The special directives (e.g. #pragma in Xilinx HLS tools[17]) available in HLS tools help in design space exploration to improve the design micro-architecture and FPGA hardware matching, but their efficient use depends on the programming abilities and experience of the developer. However, as is the case in software design, the performance profile of hardware design can help identify the performance bottlenecks and aid the developer in fine-tuning the design performance. Vivado HLS [17] and Intel HLS [7] are popular HLS tools used in industry. These HLS tools provide the overall latency of the source code along with the cycle-count at the loop or sub-function level but the cycle accounting for every line of source code is not available. Though, it is possible to relate the synthesized HDL to clock cycles

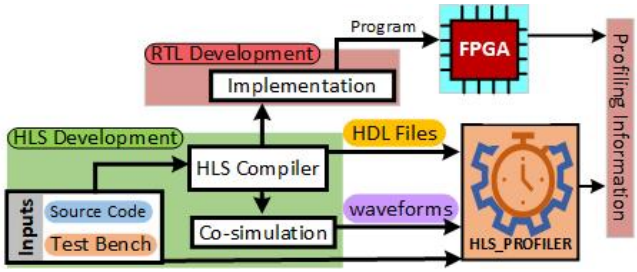


Figure 1: Representative Diagram of Profiling flow for HLS-based FPGA design.

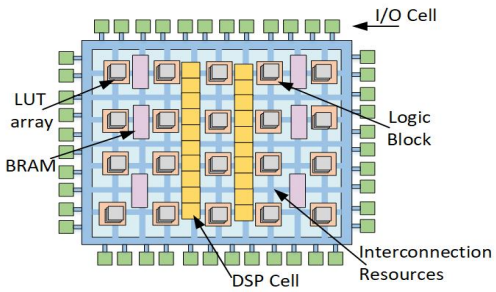


Figure 2: FPGA Internal Resources.

through waveforms, but associating the source code statements to the sythesized HDL is not straight-forward, since a single C statement can be expressed as multiple HDL signals triggering one another while being spread across multiple clock cycles.

The missing correspondence between source-code and performance information can be addressed by line-by-line profiling. This concept is not new and is available in software through tools like gprof [5]. The profilers can outline how much time has been spent in each line of code. However, this information is difficult to obtain for highly-parallel superscalar-like architectures. Additionally, it is normal for one line of source code to translate into many lines of assembly instructions, that can be scheduled with out of order parallelism. Depending on the ISA scheduling, it is possible that more than one line of code are executing in a single clock cycle. For brevity, the authors do not discuss the profiling principles adopted by gprof. [13, 18, 19] indicates that profiling tools are intrusive in nature and often introduce performance overheads. This is because of the additional profiling instructions that are added in the compiled application code to collect the required data. Apart from profiling tools, use of 'print' statements in software is a common method used to collect profile data. On hardware platforms like FPGA, HLS_Print framework [14] can be used to derive profiling information. Similar to other profiling framework, HLS_Print is intrusive in nature and introduces additional circuitry in the application. Moreover, it requires the time-consuming implementation followed by programming the FPGA and application test on hardware. On hardware platforms like FPGA, this is a step can be avoided since cycle accurate simulators are available. The design signal waveforms can be viewed in the simulator to estimate the profile of the application. The signals are part of the RTL design that is generated by the HLS compiler. However, these signals are not easily relatable to the HLS source code. This problem does not arise with applications implemented directly in RTL since the developer decides the design flow and appropriates operations based on his/her understanding of data dependencies. This is not the case in HLS since the RTL design is compiler generated with little direct correspondence to source code. However, if RTL signals could be traced back to the source code, it would have been possible to profile an HLS based application using RTL simulations without design implementation and actual application run.

Non-intrusive software simulators like Marssx86 [8] can simulate a software execution on a processor on a cycle-by-cycle basis on an x86 processor. However, they are known to execute slowly as they add virtualization layer between the application and processor hardware. This makes such software simulators impractical for industry deployment, but finds some utility in the research community working on new processor hardware designs. However, in the area of hardware design, simulation is part of standard industry process. The HLS_Profiler uses RTL signal waveforms dumped by the simulator to profile HLS designs.

To support profiling in HLS-based designs, we developed HLS_Profiler framework, an automated and non-intrusive performance profiling tool that provides a cycle-by-cycle association to every line of source code (SC) for the entire application execution time. The HLS_Profiler based approach for profiling collects waveforms, source code and HDL files from HLS developement flow and translates it to profiling information as shown in Fig. 1. Profiler framework is based on static analysis and dynamic trace, available from the HLS compilers, that is used along with associative rules to generate cycle-accurate profiles. The framework is built and tested on Xilinx HLS development tool.

We summarize the contributions of this work as follows:

- An automated and non-intrusive performance profiling tool, HLS_Profiler, for HLS based application.
- Showcase clock-wise variation of source code variables with its value and line numbers enabling fine-grained visibility.
- Developed associative rules for accurate performance reports.
- Bottleneck identification and elimination on an industrial application on session-based recommendation.
- Functional verification of the framework on MachSuite Benchmarks.

The rest of the paper is organised as follows. The HLS development flow preliminaries are discussed in Section 2 followed by proposed HLS_Profiler framework is discussed in 3. This section includes details on HLS_profiler algorithm and associative rules. Section 4 presents evaluation of HLS_profiler on Machsuite benchmarks and industrial application of recommendation system. Following this, Section 5 and Section 6 presents the related work and limitation of this work, respectively. Conclusion and future work are presented in Section 7.

2 PRELIMINARIES

In this section, we present a brief discussion on profiling and HLS development flow concepts.

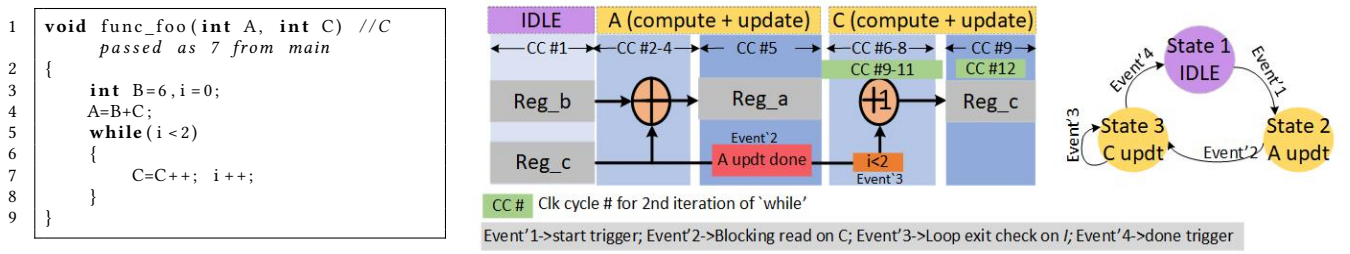


Figure 3: Sample Program (a) HLS code with its (b) synthesized RTL Design and, (c) State machine diagram.

2.1 Source code Profiling

Code profiling entails dynamic program analysis that captures code execution statistics including space (memory) or time complexity, frequency and duration of function calls *etc.* The profiling information aids in program optimization and is commonly achieved by instrumenting either the program source code or its binary executable form using profiler tools. The output of the profiler can be a statistical summary of the code with profiling data (for example #of times a line of code is executed) annotated against the source code. On the other hand, performance issues in parallel programs often depend on the time relationship of events and thus require a full trace of code execution. Profilers work on the principle of sampling and instrumentation where the former supports low-profiling granularity as compared to the latter. A sampling profiler probes the program call stack at regular intervals using interrupts. Sampling profiles are less numerically accurate and specific, but allow the target program to run at near full speed. Instrumentation technique effectively adds instructions to the target program to collect the profiling information. However, code instrumenting can cause performance changes, and may in some cases lead to inaccurate results. The instrumentation can be added manually or automatically at the source-code, intermediate-code or compiled executable level.

2.2 FPGA Resources and Datacenter Ecosystem

Modern FPGAs are offering competitive performance, low latency, sophisticated networking, high-memory bandwidth and the capability to support heterogeneous compute. Owing to these advantages, FPGAs have entered the datacenter space with CPUs and are increasingly supporting complex algorithms on its own or through hybrid systems. The FPGA device, memories (on-chip and offchip), network components (QSFP) *etc.* form an FPGA sub-system that interacts with CPUs and peripherals. The FPGA device is the programmable silicon that realizes the desired functionality. Internally, there exist a matrix of logic blocks (Look-up table (LUT) arrays, Block RAMs, DSP, Flip-flops (FFs), Multiplexers) connected through programmable interconnects and I/Os (see Fig. 2). The LUTs mimic logic gate combinations and the FFs are used as a form of storage. DSP (Digital Signal Processor) slices are used to implement signal processing functions. The Block RAMs (BRAMs) are embedded memory elements to provide on-chip storage for a set of data. Other memories include, High Bandwidth memory (HBM) and DDR that provide storage for large data and has high-bandwidth for low latency memory accesses. For instance, Alveo U280[16] offers 8GB of HBM and up to 460 GB/s bandwidth to provide high-performance and adaptable acceleration.

2.3 HLS Synthesis

In general, hardware design developers express the design task as sub-tasks and every sub-task is implemented as a module. Therefore, the overall design task contains a hierarchy of modules and module interconnection is defined through RTL datapath. Figure 3(a) and 3(b) represent a sample C code and its HLS generated RTL datapath. As part of HLS Synthesis, the HLS compiler converts application source code developed in high-level language to low-level HDL implementation and it does so mainly by following three steps. 1) Scheduling 2) Binding, and 3) Control extraction [15]. The logic operations are distributed through the clock cycles in scheduling and the number of such operations depends on the clock frequency, optimization directives and FPGA technology library. Binding assigns hardware resources for carrying out the logic operation which are scheduled using a state machine by the control extraction step. The state transitions shown in Fig. 3(c) captures the operation scheduling for code in 3(a). The three states represents idle, 'a' update and 'c' update operations triggered by events (Event' 1-4).

2.4 Static Analysis Reports

In addition to the low-level RTL implementation, the HLS compiler generates a synthesis report and Design Analysis Report (DAR) during synthesis. The synthesis report contains a summary of implemented design and presents the % of FPGA resource used, the overall latency and design violations. The design datapath, its state transitions, event scheduling and resource binding is captured in the design analysis report (DAR). Collectively, the synthesis report and DAR constitute the static analysis of the design. This is because the design aspects captured by these reports do not change during execution.

2.5 C/RTL Co-Simulation

C/RTL co-simulation uses the C test bench to automatically verify the RTL design. The HLS compiler generates the input test vectors based on the C test bench and use them for RTL simulation of the synthesized RTL. The RTL simulation output is stored as output vectors that are verified for correctness by the C test bench. The designer can review the waveform (see Fig. 4) from C/RTL co-simulation using the Wave Viewer to analyse the temporal changes of design RTL signals. The temporal changes of RTL signals can also

Figure 4: Waveform presenting temporal changes in RTL signals of sample function: *func_foo*.

Table 1: VCD dump for sample program: *func_foo*

CC#	ap_rst	A	C_o	ap_start	ap_done	ap_idle
1	1	6	X	0	0	1
2	0	6	X	0	0	1
3	0	6	X	1	0	0
4	0	6	X	1	0	0
5	0	13	X	1	0	0
6	0	X	7	1	0	0
7	0	X	7	1	0	0
8	0	8	7	1	0	0
9	0	X	8	1	0	0
10	0	X	8	1	0	0
11	0	9	8	1	0	0
12	0	9	9	1	1	0

be captured in form of VCD (Value Change Dump) report. The VCD report contains the value of RTL signals at every time-stamps for entire simulation duration. The C/RTL Co-simulation denotes the dynamic behaviour of the design and depends on run-time value of the design variables. The VCD table for *func_foo* program is shown in Table 1. The table indicates cycle-by-cycle changes of RTL design signals (*A, C_o*) and handshake signals (*ap_rst, ap_start, ap_done* and *ap_idle*). The function execution start at clock cycle (CC)# 3 when *ap_start* is asserted and finishes at CC# 12 with *ap_done* = '1'. The design signal *A* takes value *B+C* (= 13) at CC# 5. *C_o* increments are seen through CC# 9-12. It is interesting to note, that *A* changes value in CC# 8 and 11 as well. However, the functional correctness is maintained by its valid signal, *A_ap_vld*. The *A* value is valid only when *A_ap_vld* is 'HIGH'. Thus, *A* transitions are CC# 8 and 11 are invalid. The *ap_rst* and *ap_idle* indicates reset and idle conditions, respectively, of function.

Abbreviations used throughout this paper: CC- clock cycle, SC- source code, SC_Ln#- Source code line number, SC_var- variable in source code, VCD- Value Change Dump, SII- Static Information Interpreter, DTI- Dynamic Trace Interpreter, RTL- register transfer logic, HDL- hardware description language, ASC_tab- Association table, DAR- Design Analysis Report.

3 HLS_PROFILER: FRAMEWORK

The HLS compiler generates the RTL equivalent of the C source code and represent the source code variable and RTL signal association in the form of static information. The dynamic behaviour captured in form of RTL waveforms do not exhibit direct relation to C source code variables. This disconnect observed in the HLS development tools acts as a deterrent to performance fine-tuning of HLS designs. In our approach, we associate every line of C-source code (SC_Ln#) to a clock cycle (CC) through our profiling framework, called HLS_Profiler. The HLS_Profiler framework (shown in Fig. 5) uses reports generated by the HLS compiler during synthesis and co-simulation, and present a fine-grained performance profile of designs developed using HLS development tools. In addition to HLS compiler generated reports, the HLS_Profiler makes use of associative rules to suppress invalid transitions that were otherwise getting captured in the performance profile.

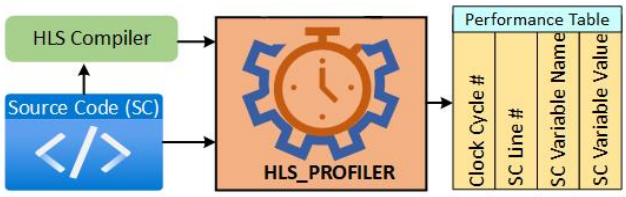


Figure 5: Representative diagram of HLS_Profiler Framework.

Figure 6 contains a sample application code. The source code has a function (*func_A*) with three variables, *a*, *b* and *c*, among them *a* and *c* updates their value during function execution. The HLS compiler generates the RTL architecture for this source code during synthesis. The performance profile shown in Fig. 6 reports temporal changes in the RTL signals for design execution time 40μs (or 4 CC) when the design operates at 100 M*Hz*. For representation purpose, we have shown only those RTL signals that are changing their value while code execution *i.e.* signals *a* and *c* with their Line #. The summary profiling information presented in tabular format in Fig. 6 indicates the SC Ln# active at every clock cycle and abstracts away the SC value related details. This table can also be represented in such a way another way such that there is only one row per source code line with the number of clock cycles it appeared in shown against it. This can be easily simplified with the second column representing the % contribution to the overall run time. However this simplification will not be detailed in this paper, due to lack of space. The state machine for *func_A* source contains three states and is shown in Fig. 7. State #1 includes reading *a*. State# 2 computes the updated values for *a* and *c* followed by write operation of updated variables. The state transition is triggered by events and contains dependence and condition checks. Event'1 is a condition check for loop exit in this case. The change of RTL signal value in a certain clock cycle is connected to its equivalent line of source code being executed. Extraction of this connection and determination of whether the corresponding SC_Ln# is active/inactive forms the basis of the HLS_Profiler algorithm.

3.1 Performance Tuning using HLS_Profiler: User View

Developer can tune the performance of the design using HLS_Profiler following simple steps as listed below:

1. Inputs: Source code, Test Bench, Synthesis Frequency and Top Function Name.

```
1  void func_A(int c)
2  {
3      int a=5;b=10;i=0;
4      while(i<2)
5      {
6          c = a^2 + b;
7          a = c;
8          i++;
9      }
10 }
```

HLS_PROFILER

Performance Profile

Time	CC#	SC Ln#	SC Var	Sig_Val
0	0	-	a	5
0	0	-	b	10
0	0	-	c	0
0	0	-	i	0
10	1	6	c	35
20	2	7	a	35
20	2	8	i	1
30	3	6	c	1235
40	4	7	a	1235
40	4	8	i	2

Summary

CC#	SC Ln#
1	6
2	7,8
3	6
4	7,8

Figure 6: HLS_Profiler framework sample input and output.

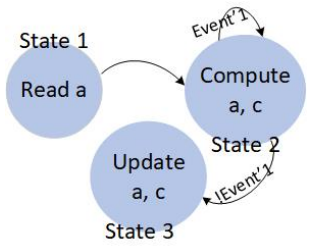


Figure 7: State machine for *func_A* source code

2. Run the HLS_Profiler script.
3. Performance profile of the source code is available in a text file.
4. Identify, optimize bottleneck and update source code with appropriate pragma directive.
5. Repeat Steps 2-4 till performance target is met.

The HLS_Profiler framework takes the C source code, C test bench, synthesis frequency and top function name as user inputs. These inputs are processed by the HLS_Profiler script to generate the Source code Performance Profile. As part of the automation, HLS_Profiler creates an HLS project followed by synthesis and co-simulation using the HLS compiler and user inputs. The framework collates the HLS compiler generated static and dynamic information to provide the performance profile for the all the design execution clock cycles.

3.2 HLS_Profiler Algorithm

The HLS_Profiler framework defines associations between line of code and clock cycle. Additionally, it highlights the source code variable and its value at every clock. We present the algorithm for defining the aforementioned associations in this section. The Profiler algorithm uses static and dynamic information made available by the HLS compiler along with special associative rules to establish accurate SC-CC mapping. The HLS_profiler flow diagram is shown in Fig. 8. The algorithm takes design source code, compiled RTL files, waveforms and synthesis report as inputs. The latter three inputs are generated using HLS compiler. The synthesis report contains state transitions, state-wise operations, its concise name and SC_vars in raw form. Concise name is a representative name for RTL/SC signals used in DAR report.

The outputs created by various blocks in HLS_Profiler framework is shown in Fig. 9 and linked by number correspondences to Fig. 8. We pre-process this information and express it in form of data structure (pp_DAR). The operations further contain its type, (out) operands, predicate and SC_Ln#. Processed DAR is used to extract the static information for the RTL signals in the Static Information Interpreter(SII) block discussed in sec. 3.2.1. SII block creates association table that contains # of state, state-wise active RTL signals with their SC variables and SC Ln#. The association table (ASC_tab) and VCD data are used by Dynamic Trace Interpreter (DTI) block. DTI performs necessary checks on the VCD temporal information to retain only valid RTL signals into the performance profile. We developed associative rules to address shortcomings of SII+DTI-alone approach. These blocks are discussed in detail in the following sections.

Algorithm 1 Static Information Interpreter

```
1:  Inputs: DAR report
2:  pp_DAR = pre-process(DAR)
3:  for i in len(pp_DAR.state) do
4:      state = pp_DAR.state[i].read();
5:      for j in len(pp_dar.state[i].op_st) do
6:          Record op_st[j].out, op_st[j].predicate, op_st[j].SC_Ln#
7:          //STEP I: Create RTL-SC_Ln# table
8:          Identify RTL_var for op_st[j].out in pp_dar.RTL_var
9:          //STEP II: Create Concise name-SC_var table
10:         Identify SC_var for op_st[j].out in pp_dar.sc_var
11:         RTL-SC_Ln#.insert({op_st[j].out,RTL_var,op_st[j].predicate,op_st[j].SC_Ln#})
12:         concise-SC_var#.write() = {op_st[j].out,SC_var};
13:     end for
14: end for
15: ASC_tab = join (RTL-SC_Ln#, concise-SC_var) on op_st.out
```

3.2.1 Static Information Interpreter (SII) Block. The synthesized RTL and source code linkages that do not change with the code execution or dynamic run are considered static correspondences. The Profiler algorithm uses Static Information Interpreter(SII) block to identify these correspondence. SII block takes processed DAR as input and generates an association table (ASC_tab: structure shown in Fig. 9). The ASC_table is obtained from Algorithm 1 in two-steps. In the first step, the RTL-SC Ln# mapping is created (Ln# 7-8 in Algorithm 1) followed by concise-SC_variable association (Ln# 10 in Algorithm 1) in the second step. The DAR contains the # of states, state-wise distribution of operations with its operands, and RTL-SC association. The states control the design execution and a group of data-independent operations can be clubbed in a single state. These

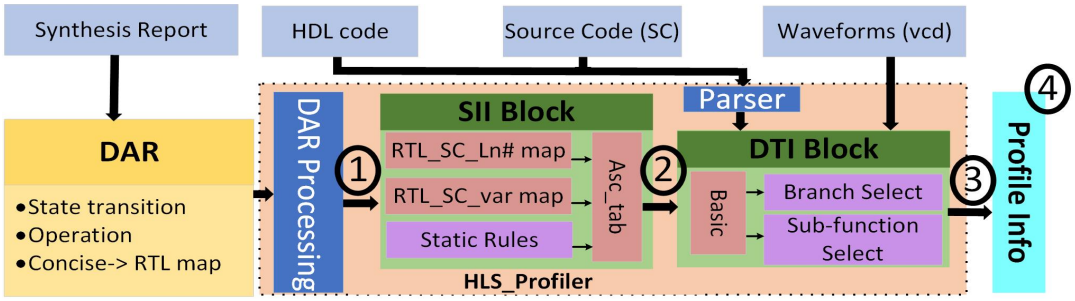


Figure 8: The HLS_profiler Algorithm block diagram.

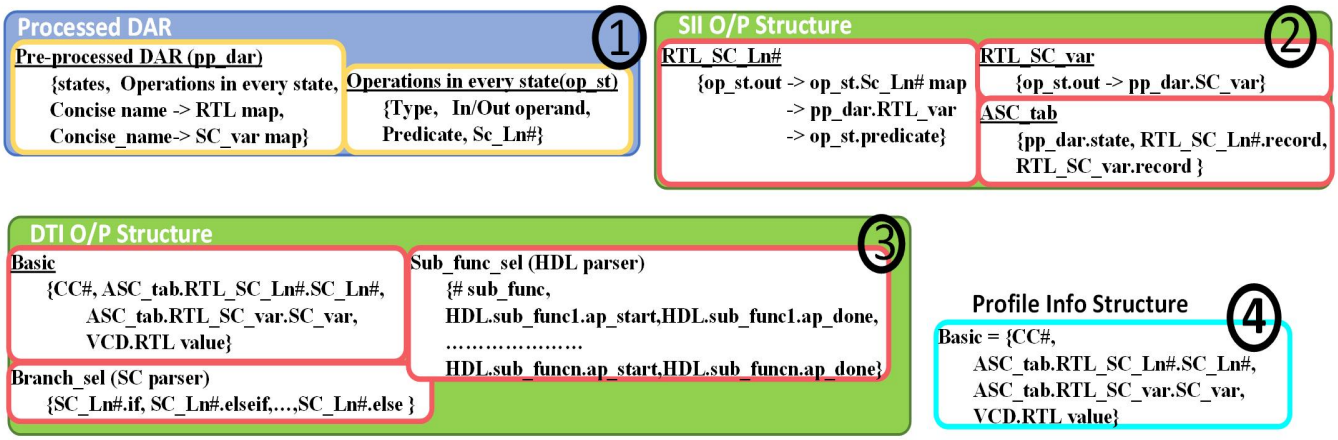


Figure 9: HLS_profiler intermediate and final output structures.

operations type can broadly be array element address calculation, read, store and compute, where compute includes mathematical (addition, multiplication, shift *etc.*) and logical (AND, OR, NOT *etc.*) functions. Every operation is characterized by its type, operands and predicate. A concise name, which is the prefix of the RTL signal, is used to indicate operands. A RTL signal that is available both as wire and reg, will share the same concise name. The mapping of concise name to RTL name is also present in DAR. Predicate denotes the pre-requisite condition for the operation and can take value 'TRUE', or operand expression. When 'TRUE' the operation gets computed unconditionally and whereas predicated containing expressions constitute conditional compute. In this case, the compute is done only when the predicate expression evaluates 'TRUE'. It should be noted, the predicate expression is static but its evaluated value depends on run-time behavior of the design.

The SC variable association with their RTL signals is available in DAR in raw format. The raw information is processed by pre-process block (shown in Fig. 8) and expressed in table format. The SII block components, RTL_SC_Ln# and RTL_SC_var are derived from the pre-processed DAR tables. The RTL_SC_Ln# structure (see Fig. 9) contains RTL name of operand with its state and predicate. The RTL_SC_var structure links operand with SC_var. These two structures are joined on operand concise name to generate the ASC_tab that contains RTL signal mapping to SC_Ln# and SC_var (Ln# 15 in Algorithm 1).

We take *func_A* code in Fig. 6 as an example to explain the structures generated in SII block. The operations scheduled in state# 2 (see Fig. 7 includes addition, multiplication, comparison and multiplexing as tabulated in Table 2. This information is present in DAR in raw form as shown in last row of the table for operation# 15. The operands are present as concise names. The predicate variable for addition and multiplication is the comparison signal at Ln# 4. The SII block algorithm converts information available in Table 2 to association table shown in Table 3. For instance, operand *a_asgn* has two RTL signals (*a_asgn_phi_fu_74 and a_asgn_reg_70*). Additionally, it has predicate 'TRUE', Ln# 7 and corresponding SC_var is

Table 2: Operations scheduled in State# 2 for *func_A* code

Op#	Type	oprnd	Pred	SC_Ln#
11	phi	i	TRUE	-
12	phi	a_asgn	TRUE	7
13	add	i_1	TRUE	8
14	icmp	icmp_ln4	TRUE	4
15	mul	mul_ln6	!icmp_ln4	6
16	add	add_ln6	!icmp_ln4	6
icmp- comparison, phi- multiplexer				
Op 15–〉 "%mul_ln6 = mul i32 %a_asgn,i32 %a_asgn"				

a. The intermediate tables: RTL-SC Ln# and concise-SC_var Tables correspondences are shown in Tables 4 and 5, respectively. The entire output would contain RTL signals, predicates, and SC Ln# correspondences for all states and is not shown here for brevity.

Algorithm 2 Dynamic Trace Interpreter (DTI)

```
1: Input: VCD and ASC_table
2: for i in len(VCD) do
3:     Single_CC_data = VCD_record[i].read();
4:     active_state = Single_CC_data.RTL_state_sig.read();
5:     active_RTL = ASC_tab.active_state.RTL_sig.read();
6:     active_RTL_Pred = ASC_tab.active_state.Pred.read();
7:     for j in len(active_RTL) do
8:         if eval(active_RTL_Pred[j]) == 'TRUE' then
9:             Add perf_record: Record[k]=active_RTL[j]; k++;
10:        end if
11:    end for
12: end for
```

3.2.2 Dynamic Trace Interpreter (DTI). The Dynamic Trace Interpreter block processes run-time information of the design and links source code with the clock cycles. The design execution is partitioned into states by the HLS compiler and state transitions are triggered by events and clock ticks. The DTI algorithm (see Algorithm 2) uses the temporal changes (signal values) in the RTL signals present in the VCD, termed as VCD_record. The algorithm tracks current execution state at a CC (Ln# 3). Once the state is identified, the algorithm filters the RTL signals active in that state from the ASC_table obtained from SII block (Ln# 4). The predicate of the filtered RTL is evaluated (Ln# 6-8). The RTL signals for which predicate evaluates 'TRUE' are added into the performance profile (Ln# 9). The performance profile entry contains CC#, SC_var, SC_Ln# and signal value. The CC# and signal value fields are obtained from dynamic analysis whereas static information provides the SC_var and SC_Ln#.

To summarize, the HLS_profiler framework associates source code line numbers and variables to clock cycles. The reports generated by the HLS compiler, are processed to create DAR tables that are further used by SII and DTI blocks, together which form the basic HLS_profiler algorithm. It should be noted that generated DAR tables may have some resemblance to LLVM IR but they are not same. The DAR tables are used to uncover source code and RTL relations.

Table 3: ASC Table output

State#	RTL Sig_name	Pred	SC_Ln#	SC_var
1	Entries related to *a* read			
2	i_phi_fu_63	TRUE	-	i
	i_reg_59	TRUE	-	i
	a_asgn_phi_fu_74	TRUE	7	a
	a_asgn_reg_70	TRUE	7	a
	i_1_fu_81	TRUE	8	i
	i_1_reg_110	TRUE	8	i
	icmp_ln4_fu_87	TRUE	4	-
	mul_ln6_fu_93	!icmp_ln4_fu_87	6	-
	add_ln6_fu_99	!icmp_ln4_fu_87	6	a
3	Entries related to *a, c* write			

Table 4: ASC_tab: RTL-SC Ln# Mapping

Oprnd (concise_name)	RTL Sig_name	Pred	SC_Ln#
i	i_phi_fu_63 i_reg_59	TRUE	-
a_asgn	a_asgn_phi_fu_74 a_asgn_reg_70	TRUE	7
i_1	i_1_fu_81 i_1_reg_110	TRUE	8
icmp_ln4	icmp_ln4_fu_87	TRUE	4
mul_ln6	mul_ln6_fu_93	!icmp_ln4_fu_87	6
add_ln6	add_ln6_fu_99	!icmp_ln4_fu_87	6

Table 5: ASC_tab: Concise-SC_var Mapping

Concise_name	SC_var
a_read	a
a_asgn	a
add_ln6	a
i	i
i_1	i

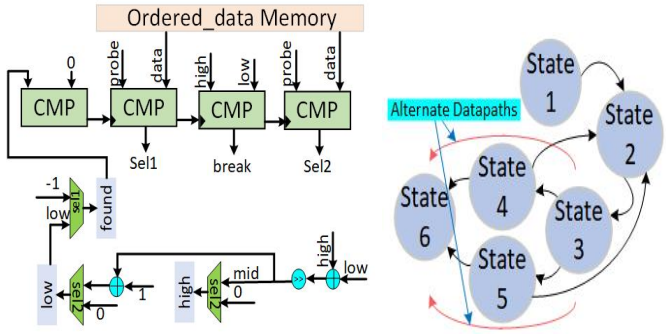
3.3 Shortcomings of basic SII+DTI algorithm

We presented the usefulness of SII and DTI blocks to obtain the performance profile in the above sections. But the associations present in SII and DTI are not absolute and can vary depending on HLS compiler behavior. Since the HLS compiler is closed source, it is not possible to alter its way of operation. However, these inconsistencies at the compiler output can add confusion to the developer's analysis of the performance profile. We present an example code (binary search) to corroborate this. In the context of this paper, the performance report correctness is estimated by metrics: False Positive (FP) and False Negative (FN). A profiler record is considered an FP when a SC_Ln# (or SC_var) denoted against a CC is incorrect. FN record indicates missing SC_Ln# (or SC_var). For an accurate performance profile, FP and FN count should be zero.

The task of the binary search program (see Fig. 10)(a) is to return the index of the input (*probe*) if found in the sorted input array (*ordered_data*). It contains a *while* loop (Ln# 7 in Fig. 10) that executes till *probe* is found or end of *ordered_data* is reached. There is branch statement (Ln# 8-18) inside *while* loop, that updates the *found* variable (Ln# 9), the search indices (Ln# 15, 16) or exits the function (Ln# 11) based on condition checks. Additionally, we see nested branches in the form of *if-else* (Ln# 14-16) pair inside the *else* statement.

The datapath in DAR for the binary search program is shown in Fig. 10(b). The RTL contains three comparators to realise if, elseif and else->if conditions. It further contains registers for *found, low, high* and a memory for *ordered_data*. Since the source contains nested branches, the comparators have an enable input that is controlled by other comparator outputs. The register write operation are controlled by comparator outputs. For instance, *found* takes the value *low* when output *sel1* is '1', otherwise it is '-1'. We observe that *mid* is input to *high* and *low* signals. The *mid+1* value is calculated and made available at *low* register input regardless of the comparator outcome. This behavior is captured in the performance profile presented in Fig. 10(d). CC# 17, Ln#16 is active even when

```
1  int bin_search(int ordered_data [8], int probe)
2  {
3    int mid;
4    int found = -1;
5    int low = 0;
6    int high = 7;
7    while (found < 0) {
8        if (probe == ordered_data[low])
9            found = low;
10       else if ( low == high)
11           break;
12       else {
13           mid = (low + high)/2;
14           if (probe <= ordered_data[mid])
15             high = mid;
16           else low = mid + 1;
17       }
18   }
19   return found;
20 }
```



Performance Profile

CC#	SC_Ln#	SC_var	SC_var value	Remark
15 (St 1)	-	found	-1	
	-	high	7	
	-	low	0	
	-	probe	3755034	
16 (St 2)	7	<cmp for while>	-	TRUE
	8	ordered_data	2133345	<probe
	8	<cmp if>	-	FALSE
	10	<cmp else if>	-	FALSE
	13	mid <compute>	3	
17 (St 3)	14	ordered_data	5114060	>probe
	14	<cmp else–⟩ if	-	TRUE
	16	low <compute>	4	
18 (St 4)	7	<cmp for while>	-	TRUE
	15	high <write>	3	
19 (St 2)	7	<cmp for while>	-	TRUE
	8	ordered_data	2133345	<probe
	8	<cmp if>	-	FALSE
	10	<cmp else if>	-	FALSE
	13	mid <compute>	1	
20 (St 3)	14	ordered_data	2133347	<probe
	14	<cmp else–⟩ if	-	FALSE
	16	low <compute>	2	
21 (St 5)	7	<cmp for while>	-	TRUE
	16	low <write>	2	
22 (St 6)	7	<cmp for while>	-	TRUE
	8	ordered_data	3755034	=probe
	8	<cmp if>	-	TRUE
	9	found <write>	2	

Figure 10: The Binary_search Program (a) C source code, (b) RTL datapath, (c) State machine, and (d) Performance Profile

<cmp else->if> evaluates 'TRUE'. According to the SC, when Ln# 14 evaluates 'TRUE', Ln# 15 is active and Ln#16 is inactive. However, functionally the RTL behaves correctly as either *high* or *low* registers (and not both) are updated in the following clock cycle. The HLS compiler pre-computes the *high* and *low* signals and schedules its computation in the same CC since there is no data dependency between them. However, their computation causes signal changes in the VCD and is captured in the performance profile obtained by basic SII+DTI algorithm.

From the binary_search case study we made two observations that challenged the performance profile correctness. Firstly, the Ln# for high update (Ln #15) was not present in the DAR, since the compiler compute *mid* at Ln #13 and treats *high* as duplicate statement. This results in a FN when 'if' evaluates 'TRUE'. Secondly, we observe that the expression *mid+1* was always executing regardless of 'if' comparison output. This computation triggered a FP transition in the temporal variations of VCD report.

3.4 Associative Rules

Based on our experience, the static analysis and dynamic trace information alone were not able to maintain correctness in the performance profile. To address this, we define a set of associative rules, that works along with the SII and DTI blocks to ensure correct performance profile output. The HLS_profiler algorithm with SII+DTI integrated with associative rules is presented as Algorithm 3.

3.4.1 Static Rules. SHIFT operation rule: HLS does not provide the RTL signal to C variables correspondence whenever the shift operation is realised in RTL. We address this in the profiler by recording Ln# of operands that have operation type ashr/trunc/bitconcatenate. Let us call this set of Ln# as shift_Ln#. The SC is parsed (see Parser block in Fig. 8) to record the SC_var present at left hand side of equality for all enteries in shift_Ln#. This gives us the required operand concise name to SC_var correspondence.

3.4.2 Sub-Function Selector. It is common for a source code to contain sub-functions. The sub-function routines gets executed when sub-function calls are encountered and remains inactive while other sections of the source code are executing. However, we observed spurious transitions from inactive sub-functions in HLS designs. This can also be attributed to the pre-emptive nature of hardware designs that compute forthcoming operations in advance. Such

Algorithm 3 HLS_Profiler Algorithm

```
1: Input: Source code (SC), HDL files, DAR and VCD data
2: ASC_tab = SII(DAR)
3: concise_SC_var_rule={Shift_op(DAR)}
4: ASC_tab = join(concise_SC_var_rule, ASC_tab) on concise name
5: Perf_record = DTI(VCD, ASC_tab)
6: for i in len(Perf_record) do
7:     Rec = Perf_record[i].read()
8:     if sub_functions in SC then
9:         for j in len(sub_func) do
10:            if Rec.SC_Ln# in sub_func[j] then
11:                if sub_func[j].ap_start != 'HIGH' then
12:                    Purge Perf_record[i]
13:                else
14:                    Retain Perf_record[i]
15:                end if
16:            end if
17:        end for
18:    end if
19:    if branch in SC then
20:        for k in len(branch) do
21:            if_rec= source_if(branch[k])
22:            if Rec.SC_Ln# in if_rec.if_cond or if_rec.elsif_cond or if_rec.else then
23:                if eval(if_rec.if_pred) != 'TRUE') and Rec.SC_Ln# in if_rec.if_cond
                   then
24:                    Purge Perf_record[i]
25:                else if eval(if_rec.elsif_pred) != 'TRUE' and Rec.SC_Ln# in
                   if_rec.elsif_cond ) then
26:                    Purge Perf_record[i]
27:                else if (Rec.SC_Ln# absent in if_rec.else) then
28:                    Purge Perf_record[i]
29:                else
30:                    Retain Perf_record[i]
31:                end if
32:            end if
33:        end for
34:    end if
35: end for
```

transitions are false positives and disturbs the correctness of performance profile report.

We use the handshake signals created by the HLS compiler to address this irregularity. The handshake signals [17] constitute of *ap_start, ap_idle, ap_done etc.* and their function is to report status of the module *i.e.* running, idle or finished. The handshake signals are created for every HDL module. The algorithm (Ln# 8-17 in Algorithm 3) tracks the current status of all the HLS modules for every clock cycle. A module is considered active between the *ap_start* and *ap_done* events. If a signal transition captured in VCD belongs to an inactive module, it is suppressed and treated as a false positive.

3.4.3 If-Else Interpreter. Source code contains branching statements such as if-else constructs depends on the run-time information to select the branch that executes. However, due to the parallel and pre-emptive mode of operation, the HLS compiler can schedule mutually exclusive branches in the same state. The functional correctness is maintained by controlling the variable update in the following state. From the performance profiler perspective, showing mutually exclusive branch Ln# in the same clock cycle is misleading and better avoided. For instance, in the binary_search code profile, Ln#16 is active at CC# 17 instead of Ln#15.

To address this, the algorithm (Ln# 18-30 in Algorithm 3) identifies the line numbers that belong to 'if'/'else if'/'then' condition from the source code by means of a parser. The if-else interpreter block creates a table, called source_if table after parsing the C source code. An example of such a table for the binary_search code in Fig 10 is Table 6. The framework can support any number of 'elseif' clauses as well as nesting levels.

Table 6: Source_if Table: Captures SC branch Line #.

if_cond	if_pred	elsif_cond	elsif_pred	else
8,9	icmp_ln8	10,11	icmp_ln10	12,18
14,15	icmp_ln14	-	-	16

4 EVALUATION

In this section, we present functional evaluation of HLS_profiler on MachSuite Banchmarks [11] and industrial application of Session-based Recommendation system [6]. The HLS compiler used for this purpose is Vitis HLS 2020.2 and Vivado HLS 2019.2.

4.1 MachSuite Benchmarks

We further evaluate the correctness on MachSuite HLS benchmarks [11]. The MachSuite benchmarks are a collection of 19 kernels developed in HLS and that exhibits frequently used tasks across many domains. Generic matrix multiplication (GEMM), Breadth First Search (BFS), sorting are few algorithms that are part of MachSuite. We tabulate the specifications of MachSuite kernels in Table 7 and observe them to be widely varied. The lines of code vary between 19 (Stencil_2D) to 660 (BFS_Queue). All the benchmarks contain loops whereas many of them have sub-functions and branch statements in them. The execution cycles ranges between 2k (encryption) to 674M in back propagation kernel. We found the performance report from these benchmarks as correct barring the cases where assumptions and limitations apply. The performance profile verification was done manually by understanding the source code behavior and control flow (sequence of line numbers) dictated by the dynamic conditional program branches were as expected for

Table 7: MachSuite Benchmarks [11] Specifications

Description	SC Lines	Source Code contains		
		Sub-Functions	Loops	Branches
AES Encryption	203	✓	✓	✓
NN training	288	✓	✓	✓
BFS (Bulk)	42	×	✓	✓
BFS (Queue)	660	✓	✓	✓
FFT (Strided)	31	×	✓	✓
FFT (Transpose)	407	✓	✓	×
GEMM (Blocked)	30	×	✓	×
GEMM (Ncubed)	20	×	✓	×
String matching	44	✓	✓	✓
MD (KNN)	58	×	✓	×
MD (Grid)	57	✓	✓	✓
DNA alignment	91	✓	✓	✓
SORT (Merge)	51	✓	✓	✓
SORT (Radix)	105	✓	✓	✓
SPMV (CRS)	22	×	✓	×
SPMV (Ellpack)	21	×	✓	×
STENCIL (2D)	19	×	✓	×
STENCIL (3D)	52	×	✓	×
HMM	64	×	✓	✓

AES - Advanced Encryption Standard, NN - Neural Network, BFS - Breadth First Search, FFT - Fast Fourier Transform, GEMM - Matrix Multiplication, MD - Molecular Dynamics, SPMV - Sparse Matrix Multiplication, HMM - Hidden Markov Model

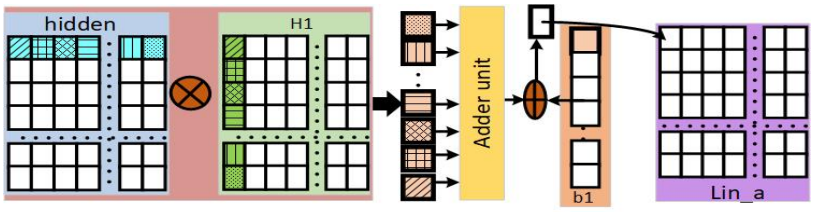


Figure 11: Compute Operations in NISER: Lin_a Block

the given test input and program state. The dynamic values of local and global variables were observed and ensured to be of the same as expected for correct program execution. It is not practical to manually verify a large number of clock cycles. However, the state transitions helped in limiting the verification method to few clock cycles and maintain confidence for the remaining cycles. We traced all possible datapaths between first and last state for all benchmarks and verified the HLS_profiler output for them. For instance, Fig. 10(c) shown two alternate paths for binary search state machine, depending on whether state 4 or state 5 is traversed. In this case, we check performance profile output on $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6$ and $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6$.

4.2 Case Study: Session Based Recommendation System (NISER)

A session-based recommendation (SR) model utilizes the information from past actions (*e.g.* item/product clicks) in a session to recommend items that a user is likely to click next. NISER (Normalized Item and Session Representations) [6] is Graph Neural Network (GNN) based SR that uses normalized representation to deal items without the popularity bias. The recommendation applications are required to make recommendations for tens of thousand of customers simultaneously and thus require designs with fast turnaround time. FPGAs are a good candidate for performance sensitive applications owing to its datapath-based processing. Internally, the NISER algorithm contains large multiple matrix multiplication operations with bias addition. We chose HLS development flow for NISER implementation on FPGA due to inherent complexity of RTL-design development for large designs.

The NISER architecture contains three blocks: *Lin_a, Lin_b* and *GNN*. *Lin_a* and *Lin_b* blocks captures graph-specific information and *GNN* block uses this output to compute the final embedding. The *Lin_a* and *Lin_b* block computes equation (1) and equation (2). Here, *hidden* is $n \times m$ matrix, *H1 (H2)* is $m \times m$ matrix and *b1 (b2)* is a column matrix with dimension $m \times 1$. n represents the # of item in session history, and is set to 10 for NISER. From equation (1) and equation (2) it is clear that operations in *Lin_a* and *Lin_b* blocks are essentially the same and only difference is in the operands. Also, it should be noted that there is no data dependence between these two blocks.

$$Lin\_a = hidden \cdot H1 + b1 \quad (1)$$

$$Lin\_b = hidden \cdot H2 + b2 \quad (2)$$

Equation (1) term $hidden \cdot H1$ is coded as three nested for loops: *outer*, *middle* and *inner* in the HLS. *Outer* loops over all rows of *hidden*, middle loops over all columns of *H1* and *inner* loop contains multiplication of row elements of *hidden* with corresponding column elements of *H1*. For instance, first row of hidden (cyan fill cells in Fig. 11) and first column of H1 (green fill cells in Fig. 11) are used to compute first element (row=0, col=0) of lin_a matrix. m product terms are generated by multiplying corresponding elements of active row and column that are accumulated in adder unit. The adder unit result is further added with bias term to generate (0,0) element of *Lin_a* matrix. In the code excerpt presented in Fig. 12, Ln# 3-6 indicates *inner* loop multiplication operation. The multiplier outputs accumulation is done in Ln# 7-11 and addition with bias term is indicated as Ln# 12. This operates inside *outer* and *middle* loops placed at Ln# 1 and 2, respectively.

We now discuss translation of code excerpt presented in Fig. 12 to RTL design. The arrays *hidden*, *H1*, *local_row*, *b1* and *Lin_a* are implemented as BRAM memory elements. The HLS compiler creates a dual-port BRAM memory by default. This means that only two BRAM elements can be accessed (written to/read from) simultaneously. Additionally, on a single port one type of access (write/read) is supported at a time. The depth and width of the BRAM is determined by # of array elements and its datatype, respectively. The multiplication, accumulation and addition are implemented as dedicated pipelined DSP units. The loop exit condition and increment are implemented as comparison and addition operation, respectively. In terms of resources, both operations are realised in LUTs and registers.

To analyse the performance bottleneck in *Lin_a* block, we exposed its HLS code to HLS_Profiler and estimated the SC_Ln-wise latency. The multiplication at Ln# 7 is part of *outer: for* loop and is implemented in hardware as completely pipe-lined DSP taking 2 CC. The accumulate at Ln# 10 can not be fully pipe-lined because of data-dependency (*result* variable) and takes 3 CC. The addition at Ln# 12 take 5 clock cycles and resides in *middle: for* loop. In this case, eliminating the bottleneck from level-3 *outer:for* loop will bring greater performance gain. We apply *array_partition+loop_unroll* pragma directives on *inner_mul* loop. This reduces its latency from 300 CC to 2 CC. In effect, the *array_partition* pragma eliminates the port limit on BRAMs and all elements in the array can be accessed at once. Due to this optimization, the end-to-end latency for *Lin_a* block is improved by 2.3×.

5 RELATED WORK

The manual approach for performance tuning needs a balanced use of pragma directives and deep understanding of the design micro-architecture. The choice of pragmas highly depend on the interactions of design sub-modules and its datapath. This makes the

```
1  outer: for (int i = 0; i < a_row; i++)
2      middle: (int j = 0; j < b_col; j++)
3          inner_mul:
4              for (int k1 = 0; k1 < a_col; k1++) {
5                  local_row[k1]=hidden[i][k1]*H1[k1][j];
6              }
7          inner_add:
8              float result=0;
9              for (int k2 = 0; k2 < a_col; k2++) {
10                 result +=local_row[k2];
11             }
12         lin_a[i][j] = result + b1[j] ;
13     }
14 }
```

Loop-wise Latency (in CC)			
[redacted]	Vanilla	Optimized	Speed-up
outer	708020	306020	2.3×
middle	70800	30600	2.3×
inner_mul	300	2	150×
inner_add	400	300	1.3×

SC Line-wise latency (in CC)			
[redacted]	Ln# 5	Ln# 10	Ln# 12
	2	3	5

Figure 12: Session-based Recommendation Application (a) source code, and (b) performance profile and gain

manual tuning task time-consuming and challenging since most of the time the pragma positioning and choices are made based on trial and run approach. On the other hand, the DSE techniques search for pareto-optimal solutions while optimizing conflicting design parameters such as area, performance, power *etc.* Many HLS DSE tools have been proposed in literature [10, 12, 20, 21, 23] that finds optimal designs by automatically making best use of pragmas. These tools use resource and performance estimators to evaluate micro-architectures and leverage similarity between source codes to speed up the exploration process. At present, the performance estimators used by DSE tools provide end-to-end, loop-level or function-level latencies. This propels the DSE to apply similar optimization pragma to all loops/sub-functions which may not be optimal. Additionally, it is difficult to get a reliable performance estimate for loops with variable bounds.

Researchers have focused their attention to develop dedicated HLS performance estimator tools [2–4, 9, 22] that can give insights on performance bottlenecks, stall rate, stall cause *etc.* These estimators often limit themselves to simple loop topologies and limited pragma use which makes them unreliable for large designs with complex datapaths. This motivates us to develop a cycle-accurate, fine-grained performance profiling framework that is non-intrusive and provides an end-to-end profile of the design. Such profiling tool can help the designer/DSE tool to quickly identify the performance bottlenecks and have a guided approach towards tuning it.

5.1 Comparison with State-of-the-Art

HLScope+ [3, 4] uses code instrumentation and analytical modelling to improve the accuracy of the Vivado HLS simulator. It is two orders of magnitude faster than the whole synthesis process but requires HLS simulation for each design point. Aladdin [12] and Lin-analyzer [22] can also be used to provide cycle estimate for programs with dynamic behavior since they utilize the instruction trace generated in C simulation. Moreover, these works focus on providing analysis for efficient exploration of possible design points. [20, 21] rely on an estimation of performance and resource requirement of a given optimization. While mandating very few synthesis runs, such strategies struggle when coping with multiple, interdependent optimizations. Hence, they are often limited to capturing the effect of only a few directives. HLS_profiler aligns toward evaluating an actual design that is synthesized by an HLS tool and provides accurate performance profile.

6 ASSUMPTIONS AND LIMITATIONS

In this section we highlight assumptions and limitations in the current state of the tool. The framework works on the assumption that the source code is written following good coding practices and contains single statement at every line. The limitations are listed below.

- Performance profile verification on source codes with optimization directives is not exhaustive and is limited to few pragmas *viz. array_partition* and *loop_unroll*.
- Verification has been manually done. Control flow of source code was hand simulated. It will be ideal to devise an automated method to verify the profiler results and the variable values reported by the HLS_profiler. One possible method will be to use gprof to trace for a CPU execution and using that to verify the control flow.
- As the source code line execution may overlap many clock cycles it is often observed that many source lines are active in a given cycle. This could be a result of the FPGA scheduling instructions in parallel. If we sum up the clock cycles spent in each line of source code the total will exceed the overall program execution time. This is because the HLS_profiler works bottom up - it works its way from the RTL signal waveforms to the source code. This is quite different from other profilers which base its measurements on inserting timestamps like it is done in software. This could be a challenge for some developers, but it will be possible to create scripts which can parse the profile to identify longest running operations in case when many operations are scheduled concurrently. This may not be easy because it is possible that all operations did not get schedule for execution at the same time. Getting a perfect profile wherein total latency of all the source lines is equal to the program latency is being addressed but not complete at the time of writing this paper.
- The HLS_profiler framework is tested on few recent versions of HLS compiler including Vivado and Vitis. However, this doesn't guarantee it will work for future version of compilers, since the profile generation depends on compiler outputs, especially if compiler outputs change significantly. Nevertheless, the framework would provide sufficient information to derive useful insights about performance profile.

There are cases where some of the source code line numbers do not show up in the profile even if they are effective at a given clock cycle. They include

- Initialization statements - the registers which hold the variables are initialized upon reset (startup) even before any function starts execution
- Line numbers for source code statements that use HLS library functions such as sin or cos function are not captured by the framework. This is because the HLS tool implements *sin* or *cos* function of the variable by instantiating hardware macros from the respective libraries.
- The profiler is not able to capture the correct line numbers of expressions that are duplicates of some other expressions, because those expressions get optimized by the compiler.[1]
- The profiler is unable to capture the value of some of the source code variables, especially those of pointers.

The authors believe that using methodology of discovering rule associations described earlier it would be possible to uncover relationships that will expose the executions of such source code lines in the profile.

7 CONCLUSION AND FUTURE WORK

Performance profile is an important instrument to efficiently explore the design space and systematically improve its performance. The challenges in relating the HLS code to the synthesized RTL acts as a deterrent in analyzing the implementation performance, when it comes to non-intrusive analysis. Moreover, control and branching statements add to the difficulty in performance evaluation based only on static information. We joined the static information with hardware signal waveforms to come up with methods that could be used to profile the application actual execution on the FPGA. We formulated useful association rules that could help relate hardware signals and waveforms to the source code variables and line numbers. These discoveries incorporated into the HLS_profiler were enough to correctly profile all the diverse applications in the Machsuite Benchmarks. We were also able to demonstrate a practical use case wherein we could profile an industrial application making deep learning based recommendations. The profiling analysis enabled us to make changes to arrive at a faster implementation. We have also listed the limitations of the tool and its testing which we intend to improve upon in future. In addition to profiling, we would address area and energy consumption details in future work that designer can use to make informed decisions for HLS designs.

REFERENCES

[1] Donald G. Bailey. 2015. The Advantages and Limitations of High Level Synthesis for FPGA Based Image Processing. In *Proceedings of the 9th International Conference on Distributed Smart Cameras* (Seville, Spain) *(ICDSC '15)*. Association for Computing Machinery, New York, NY, USA, 134–139. https://doi.org/10.1145/2789116.2789145

[2] André Bannwart Perina, Jürgen Becker, and Vanderlei Bonato. 2019. Lina: Timing-Constrained High-Level Synthesis Performance Estimator for Fast DSE. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. 343–346. https://doi.org/10.1109/ICFPT47387.2019.00063

[3] Young-Kyu Choi and Jason Cong. 2017. HLScope: High-Level Performance Debugging for FPGA Designs. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 125–128. https://doi.org/10.1109/FCCM.2017.44

[4] Young-kyu Choi, Peng Zhang, Peng Li, and Jason Cong. 2017. HLScope+,: Fast and accurate performance estimation for FPGA HLS. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 691–698. https://doi.org/10.1109/ICCAD.2017.8203844

[5] Jay Fenlason and Richard Stallman. 1998. *The GNU Profiler.* https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html

[6] Priyanka Gupta, Diksha Garg, Pankaj Malhotra, Lovekesh Vig, and Gautam M. Shroff. 2019. NISER: Normalized Item and Session Representations to Handle Popularity Bias. *arXiv: Information Retrieval* (2019).

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

[8] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSS: A full system simulator for multicore x86 CPUs. *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)* (2011), 1050–1055.

[9] André Bannwart Perina, Jürgen Becker, and Vanderlei Bonato. 2019. ProfCounter: Line-Level Cycle Counter for Xilinx OpenCL High-Level Synthesis. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 618–621. https://doi.org/10.1109/ICECS46596.2019.8964669

[10] Nam Khanh Pham, Amit Kumar Singh, Akash Kumar, and Mi Mi Aung Khin. 2015. Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 157–162.

[11] 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.

[12] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 97–108. https://doi.org/10.1109/ISCA.2014.6853196

[13] Amitabh Srivastava and Alan Eustace. 1994. ATOM: A System for Building Customized Program Analysis Tools. *SIGPLAN Not.* 29, 6 (June 1994), 196–205. https://doi.org/10.1145/773473.178260

[14] Nupur Sumeet and Manoj Nambiar. 2021. HLS_PRINT: High Performance Logging Framework on FPGA. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (Virtual Event, France) *(ICPE '21)*. Association for Computing Machinery, New York, NY, USA, 145–152. https://doi.org/10.1145/3427921.3450238

[15] Xilinx. 2018. *Vivado Design Suite User Guide: High-Level Synthesis.* Retrieved June 10, 2020 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf

[16] Xilinx. 2019. *Adaptable Accelerator Cards for Data Center Workloads.* Retrieved June 15, 2020 from https://www.xilinx.com/publications/product-briefs/alveo-u280-product-brief.pdf

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

[18] Karim Yaghmour and Michel Dagenais. 2000. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *USENIX Annual Technical Conference, General Track.*

[19] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. 1996. Performance Analysis Using the MIPS R10000 Performance Counters. In *Supercomputing '96:Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*. 16–16. https://doi.org/10.1109/SUPERC.1996.183522

[20] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 430–437. https://doi.org/10.1109/ICCAD.2017.8203809

[21] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. 2020. Performance Modeling and Directives Optimization for High-Level Synthesis on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 7 (2020), 1428–1441. https://doi.org/10.1109/TCAD.2019.2912916

[22] Guanwen Zhong, Alok Prakash, Yun Liang, Tulika Mitra, and Smail Niar. 2016. Lin-Analyzer: A high-level performance analysis tool for FPGA-based accelerators. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2897937.2898040

[23] Guanwen Zhong, Vanchinathan Venkataramani, Yun Liang, Tulika Mitra, and Smail Niar. 2014. Design space exploration of multiple loops on FPGAs using high level synthesis. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. 456–463. https://doi.org/10.1109/ICCD.2014.6974719

[1] **This is currently a work in progress and addressed by typecasting source code and leveraging scheduling information.**