HLScope: High-Level Performance Debugging for FPGA Designs

Young-kyu Choi and Jason Cong
Computer Science Department, University of California, Los Angeles
{ykchoi, cong}@cs.ucla.edu

***Abstract*—In their quest for further optimization, field-programmable gate array (FPGA) designers often spend considerable time trying to identify the performance bottleneck in a current design. But since FPGAs do not have built-in high-level probes for performance analysis, manual effort is required to insert custom hardware monitors. This, however, is a time-consuming process which calls for automation. Previous work automates the process of inserting hardware monitors into the communication channels or the finite-state machine, but the instrumentation is applied in low-level hardware description languages (HDL) which limits the comprehensibility in identifying the root cause of stalls. Instead, we propose a performance debugging methodology based on high-level synthesis (HLS). High-level analysis allows tracing the cause of stalls on a function or loop level, which provides a more intuitive feedback that can be used to pinpoint the performance bottleneck. In this paper we propose HLScope, a source-to-source transformation framework based on Vivado HLS for automated performance analysis. We present a method for analyzing the information collected from the software simulation to estimate the stall rate and its cause without the need for FPGA bitstream generation. For detailed analysis, an in-FPGA analysis method is proposed that can be natively integrated into the HLS environment. Experiments show that the parameter extraction from the simulation process is orders of magnitude faster than bitstream generation, with a 2.2% cycle difference on average. In-FPGA flow consumes only about 170 LUTs and a BRAM per monitored module and provides cycle-accurate results.**

I. Introduction

When the initial field-programmable gate array (FPGA) implementation does not meet the required performance, designers often iterate the process of identifying the bottleneck in the current design and finding an optimization to fix the problem. However, unlike CPU and GPU designers who can use built-in hardware counters with established tools like VTune [7] and NSight [11] for the performance debugging process, FPGA designers usually insert hardware monitors into their design manually. For a large design, this task could take a very long time; this is a problem that calls for automation.

Previous work on automating the performance debugging process relies on instrumenting hardware monitors into DRAM/inter-module FIFO communication channels [3], [4], [6], [8], [9], [13] or into the finite-state machine of a loop pipeline [3], [13] to measure their active/idle cycle ratios. However, their instrumentation is performed from the viewpoint of individual module with low-level hardware description language (HDL). Such limited scope makes it difficult to analyze the root cause for the stall. Instead, we propose a performance debugging methodology based on high-level synthesis (HLS). High-level analysis allows tracing the cause of stalls on a function or loop level, which provides more intuitive feedback to the programmer to pinpoint the bottleneck of an FPGA design to further optimize the design.

However, challenges exist for HLS-based performance debugging. The first problem is that generating bitstream after inserting performance monitoring logic takes many hours. The second problem is that HLS abstracts hardware cycle information away from the user. Mixed HLS-HDL flow [3] can be used to extract cycle, but such flow complicates the integration process in HLS tools. For example, Intel's OpenCL HLS tool [6] does not allow mixed HLS-HDL flow, and Xilinx's SDAccel [14] (based on Vivado HLS [15]) only recently allowed it in their latest version.

To address these two challenges, we propose HLScope, a performance debugging framework based on HLS for FPGA. HLScope can interpret the information collected from HLS software simulation. Based on the analysis, our tool provides the stall rate and its cause, such as external DRAM access, synchronization, and dependency stalls on a module level. This can guide programmers to focus their attention on inefficient modules. This flow is several orders of magnitude faster than generating bitstream after hardware monitor insertion. The second contribution is the cycle-accurate in-FPGA monitoring in pure HLS code. Using non-blocking FIFO access and pipelining, we propose a source-to-source (S2S) transformation method that can be expressed in HLS without the need for mixed HLS-HDL flow. The code instrumentation for parameter extraction is automated, as we will be demonstrating with the quicksort example.

II. Performance Debugging Framework

A. Overall Framework and Parameters

HLScope framework is shown in Fig. 1. Using the APIs in the ROSE compiler infrastructure [12], our tool first analyzes the input HLS code to find the dependency between modules. The modules are classified as being executed in serial or in parallel and stored in a hierarchical structure. Next, the number of cycles for each module is measured using the Vivado HLS software simulation with a real input testbench. Our tool automatically makes source-to-source transformation to include the hardware cycle model in the software source code (Section II-B). In addition to cycle estimation, we also record the number of DRAM transactions in bytes for further analysis.

Using the module structure and cycle information, we mark each module as to whether it is or is not on the critical path of overall performance (PCP). The serially executing modules are classified as 'yes,' and the most time-consuming among parallel executing modules are also marked 'yes.' The rest are classified as 'no.'

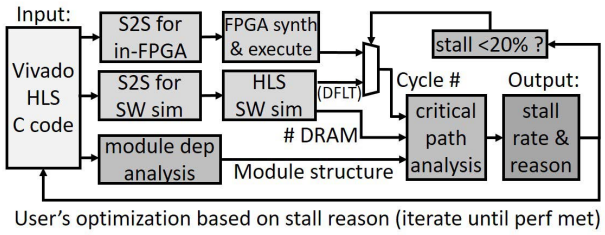

Fig. 1: HLScope overall framework

Based on the critical path, we compute the stall rate as the number of cycles of a module divided by the longest critical path. Next, we provide the reason for the stall and the name of the module that is causing the stall. If a module is waiting for data from another module, the stall is classified as a dependency stall. If a module is waiting for other parallel-executing modules to finish, the stall is classified as a synchronization stall. A module could have multiple stall reasons depending on its place in HLS module hierarchy.

Since the accuracy of software simulation is within 2–7% on average (Section IV-C), it is possible that softeware-based analysis might provide misleading information if modules executed in parallel have similar cycle numbers. This will result in the stall rate of modules in the critical path having low value (we set threshold to 20%). In this case, our tool will recommend going through the in-FPGA flow for accurate analysis that can replace the cycle information from the software simulation flow. This flow is explained in Section II-C.

Next, we determine DRAM bandwidth (DBW). The number of transferred bytes is from the software simulation. The time taken is determined either in simulation flow or in-FPGA flow. We also provide the aggregate DRAM bandwidth (ADBW) among all modules executed in parallel. We provide LUT/BRAM/DSP numbers from the HLS synthesis report that can be used to determine the importance of each module.

Based on the reason for stall provided by HLScope, the programmer can decide which module to focus his/her attention on for further optimization. An example performance debugging session will be presented in Section III.

B. Cycle Estimation Based on Software Simulation

To illustrate that we can obtain a good cycle estimate even for input-dependent applications, we use the quicksort example [5]. The synthesis report by Vivado HLS [15] cannot provide the cycle estimate due to the existence of dynamic execution path and undetermined loop trip count (TC). The tool will only provide the initiation interval (II, input rate of the loops) and iteration latency (IL, latency per one loop iteration).

We can obtain the actual TC by inserting a simple counter statement, as shown in Fig. 2. Regardless of the existence of undetermined loop bound or conditional break statement, TC can be correctly estimated. Then the loop cycle estimation [10] code is inserted after the loop based on the run-time acquired TC. Finally, the cycle estimate for Loop 2 is hierarchically added into its parent loop (Loop 1) cycle estimate.

Although details are omitted, we also apply techniques to estimate cycles for dynamic execution path and the loops that are not perfectly nested. For DRAM modeling, the access cycle is estimated as t=$DSIZE/DBW$ +$DLAT$, where the latency $DLAT$=580ns and bandwidth $DBW = 9.05GB/s$ (obtained using the method in [2]), on ADM-PCIE-7V3 [1].

```
int loop_1_cycle = 0;
while (L<R) {  // Loop 1    // Swap until L ptr & R ptr meets
  ......
  int loop_2_cycle = 0;
  int loop_2_TC=0;
  while (arr[R]>=piv && L<R){ // Loop 2   // Decrement R ptr
    #pragma HLS pipeline      //until an element less than
    R--;                      //pivot is found or L & R meets
    loop_2_TC++;
  }
  loop_2_cycle = (loop_2_TC-1) * loop_2_II + loop_2_IL;
  loop_1_cycle += loop_2_cycle;
  ......                      // Swap arr[L] and arr[R]
}
```

Fig. 2: Code instrumentation for software simulation to find dynamic loop bound in quicksort [5]. Instrumented code is in bold.

```
void qsort_top_new(...){
#pragma HLS dataflow //mntr and mods run in parallel
  stream<bool> p0, p1, p2, p_endmtr;  // FIFO probes
  qsort_top_old(..., p0, p1, p2, p_endmtr);
  monitor_3p(p0, p1, p2, p_endmtr);
}
```

(a) Connection from modules under analysis to monitor logic

```
void qsort_compute( ..., stream<bool>&p0 ){
  p0.write(1); // module start signal through probe
  ... ...
  p0.write(0); // module end signal
}
```

(b) Code instrumentation to signal monitor for module start/end

```
void monitor_3p(stream<bool>&p0, p1, p2, p_endmtr){
  bool endmtr=0; int p0_start=0; ...; int cycle=0;
  while(endmtr == 0){
#pragma HLS PIPELINE II=1 //loops once per HW cycle
    . . .
    // start/end probe processing (please see (d))
    . . .
    bool endmtr_data = 1;     //monitor termination
    if ((p_cycle.read_nb(endmtr_data)) == 1) {
      if (endmtr_data == 0) { endmtr = 1; }
    }
    cycle += 1;   //"cycle" becomes actual HW cycle
} }
```

(c) Monitor logic structure & termination probe processing

```
  bool p0_data = 1;
  if ((p0.read_nb(p0_data)) == 1){ //non-blk read
    if (p0_data == 0) {            //success
      dbg_mem0 += cycle - p0_start;
    }                              // record module end
    else {
      p0_start = cycle;     // record module start
  } }
```

(d) Start/end probe processing in monitor logic

Fig. 3: Code instrumentation for in-FPGA monitoring. Three submodules of interest (load(), qsort_comp(), and store()) exist in qsort_top_old().

C. Cycle Extraction Based on In-FPGA Monitoring

An in-FPGA monitoring flow can be used to extract cycle information more accurately than the simulation-based flow, at the cost of spending time on generating the FPGA bitstream. For easy integration with HLS-based synthesis flow [14], we would like the monitoring logic to be expressed in pure HLS code. Then the biggest challenge is to extract cycle information, which is hidden from programmers in HLS. To

```
for( int i = 0 ; i < batch_num ; i++ ){      ||SW_cyc|PCP |LUT |BRAM|DSP|Stall|DBW |ADBW|Reason for stall              ||
  load(dram,lmem,n_per_batch,i);             ||145k  |Yes |481 |0   |4  |98.2 |724M|724M|comp(96.5,dep),store(1.7,dep)||
  qsort_comp(lmem, n_per_batch);             ||7.75M |Yes |2538|3   |0  |3.5  |0   |0   |load(1.8,dep),store(1.7,dep) ||
  store(dram,lmem,n_per_batch,i);            ||140k  |Yes |445 |0   |4  |98.3 |748M|748M|comp(96.5,dep),load(1.8,dep) ||
}
```

Fig. 4: Performance debugging parameters collected from the initial unoptimized version of quicksort. Parameter derived from the software simulation (SW_cyc) result. (batch_num=128, n_per_batch=1024, 'dram': external DRAM port, 'lmem': local BRAM).

```
float local_data[UNROLL_FACTOR]; //partitioned BRAM for parallel access, UNROLL_FACTOR=32
#pragma HLS ARRAY_PARTITION variable="lmem" complete
                                    ||SW_cyc|PCP |LUT |BRAM|DSP|Stall|DBW |ADBW|Reason for stall                          ||
for( int i = 0 ; i < batch_num ; i++ ){     |    |    |    |   |     |    |    |                                         ||
  load( dram, lmem, ..);            ||132k  |Yes |3017|0   |4  |74.8 |797M|797M|comp27(49.5,dep),store(25.2,dep)         ||
  for( int j = 0 ; j < UNROLL_FACTOR ; j++ ){    |    |    |   |     |    |    |                                         ||
#pragma HLS unroll // 32 duplicated compute PEs  |    |    |   |     |    |    |                                         ||
    qsort_comp(lmem[j]..); //PE0    ||247k  |No  |2538|3   |0  |52.6 |0   |0   |ld(25.2,dep),comp27(2.2,sync),st(25.2,dep)||
    . . .                  //PE7    ||231k  |No  |2538|3   |0  |55.7 |0   |0   |ld(25.2,dep),comp27(5.3,sync),st(25.2,dep)||
    . . .                  //PE27   ||258k  |Yes |2538|3   |0  |50.5 |0   |0   |ld(25.2,dep),st(25.2,dep)                 ||
  } . . . . . .                     ||      |    |    |    |   |     |    |    |                                         ||
  store( dram, lmem, ..);           ||131k  |Yes |5809|0   |4  |74.8 |798M|798M|comp27(49.5,dep),load(25.2,dep)          ||
}
```

Fig. 5: Performance debugging parameters after unrolling qsort_comp() function 32 times. Some PEs not shown for brevity.

```
for( int i = 0 ; i < batch_num ; i++ ){
  if( i%2 == 0 ){ //in even i, lmem0 for memory access and lmem1 for compute
                                                ||HW_cyc|PCP |LUT |BRAM|DSP|Stall|DBW |ADBW|Reason for stall    ||
    loadstore(dram,lmem0,..);                   ||299k  |Yes |9145|0   |8  |0.0  |701M|701M|                   ||
    for( int j = 0 ; j < UNROLL_FACTOR ; j++ ){ ||      |    |    |    |   |     |    |    |                   ||
#pragma HLS unroll // 32 duplicated compute PEs ||      |    |    |    |   |     |    |    |                   ||
      qsort_comp(lmem1[j],..); //PE0            ||266k  |No  |2538|3   |0  |11.0 |0   |701M|loadstore(11.0,sync)||
    . . .                      //PE7            ||250k  |No  |2538|3   |0  |16.4 |0   |701M|loadstore(16.4,sync)||
    . . .                      //PE27           ||277k  |No  |2538|3   |0  |7.4  |0   |701M|loadstore(7.4,sync) ||
   else{ . . . }  //in odd i, lmem1 for memory access and lmem0 for compute    . . . . . .
} }
```

Fig. 6: Performance debugging parameters after applying double buffering optimization. Cycle derived from in-FPGA monitoring (HW_cyc) result.

solve this problem, we propose a technique of using non-blocking FIFO read and pipelining.

HLScope instruments two types of probes: 'pX' probes to signal start and end of module execution ('1' for start and '0' for end) and 'p_endmtr' to signal monitor termination. It also inserts 'dbg_memX' global variable to record cycle and a monitor logic for overall processing.

A sample code instrumentation is given in Fig. 3 for quicksort. HLScope inserts a monitoring logic monitor_3p() to run in parallel with qsort_top_old() which contains submodules under analysis (Fig. 3a). For each submodule, 'pX' probe sends 1/0 to signal module start/end (Fig. 3b). These 'pX' probes are connected to the monitor through FIFO (Fig. 3a). For monitor termination, 'p_endmtr' sends a value of 0 at the end of qsort_top_old() module (code omitted).

In the monitoring logic, variable 'cycle' corresponds to the actual hardware cycle. This is possible since the loop is pipelined to II=1, and all FIFO reads are declared non-blocking (Fig. 3c). Then the loop can run continuously regardless of the input. Note that 'cycle' is incremented by one for each loop iteration.

Based on 'cycle' variable, we subtract the cycle obtained at the start of submodule 0 execution from the cycle obtained at the end, and accumulate to 'dbg_mem0' (Fig. 3d). The same is done for submodule 1 and 2. After the 'p_endmtr' has been processed for program termination (Fig. 3c), 'dbg_memX' content is written to DRAM for analysis (code omitted).

III. Performance Debugging for Quicksort

In this section we will demonstrate performance optimization steps for the quicksort example based on HLScope. We assume that we have 128 sets of 1024 single-precision floating-point numbers to be sorted.

Fig. 4 lists the performance debugging parameters collected from initial unoptimized quicksort. Since this is the first debugging iteration, the parameters are derived from the software simulation. The most obvious problem that we can identify from the report is that qsort_comp() takes most (96.5%) of the execution time, and probably should be the target for optimization. Note that the stall rate is very high (98.2%/98.3%) for load() and store(), but does not cause significant problems since the LUT and DSP usage is small (481/445 and 4/4).

Based on the analysis from the initial version, we apply unrolling on the compute PEs. The result is shown in Fig. 5. We can confirm that the qsort_comp() function indeed takes a considerably shorter time—from 7.75M cycles to 231–258k cycles. For the nodes on the time-critical path (load(), qsort_comp() PE 27, store()), however, the analysis shows that the proportion of load() and store() increased to 25.2%, respectively. This suggests that memory access has now become a major stall reason. A hint for solution can be found in the aggregate DRAM BW. During qsort_comp(), the ADBW is 0, which means that DRAM is not being utilized at all. This suggests that modules that do use the DRAM, load() and store(), can probably be overlapped in qsort_comp().

For memory and compute overlapping, we perform double buffering optimization. For simplicity of design, we combined the load() and store() modules into one. The result in Fig. 6 shows that there is some DRAM BW transaction while the qsort_comp() module is being executed (ADBW=701M). Also, there are no more dependency stalls as reasons for a stall. These two factors suggest that parallelization is properly taking place. Also, the fact that the stall rate has decreased drastically from the initial version (98.2/3.5/98.3 $\rightarrow$ 0.0/11.0/16.4/7.4) suggests that overall efficiency of the design has improved significantly.

After double buffering, the stall rate becomes low ($<$20%) for all modules with PCP. Thus, we have switched to in-FPGA analysis. The result is similar but with higher accuracy. The report indicates that the bottleneck is now loadstore(), since all other modules point to this module as their reason for stall, and only loadstore()'s PCP is 'yes.' Since the DRAM BW reported that (701MB/s) is far less than ideal BW (9.05GB/s), we can perform some DRAM access optimization for further improvement.

IV. Experimental Results

A. Experimental Setup

For our evaluation platform, we use the Alpha Data ADM-PCIE-7V3 board [1] that has Xilinx's Virtex 7 690T FPGA. For the FPGA synthesis, we use the Xilinx SDAccel 2016.2 [14] and Vivado HLS 2016.3 [15] software tools.

B. Overhead for Performance Debugging

1) Time Overhead: The source-to-source transformation for software simulation-based flow, module dependency analysis, critical path analysis, and stall reason analysis takes 27–51 seconds (Table I). Also, simulation of the instrumented code requires extra overhead that is proportional to the original software simulation time. On average, it was 31%. For the in-FPGA flow of the same applications, the debugging flow takes 13–33 seconds. However, it takes several hours for synthesis. Thus the software flow is several orders of magnitude faster, which makes it appropriate for rapid analysis.

TABLE I: Overhead of software simulation flow for various versions of quicksort. Consists of code instrumentation and additional software simulation time. (unit:s)

	SW Dbg	SW Sim Unmod	Instr SW Sim Est	In-FPGA Dbg	Bitstr Gen
unopt	27	0.026	0.038 (1.46X)	13	1h27m
unroll	37	0.024	0.034 (1.42X)	22	2h4m
dubbuf	51	0.025	0.034 (1.35X)	33	1h55m
dramopt	51	0.280	0.287 (1.02X)	33	7h58m
AVG		(1.0X)	(1.31X)		

2) Logic Overhead: The in-FPGA flow has logic overhead for the performance monitor insertion (Table II). The logic consumption increases with the number of modules to be monitored at the rate of approximately 170 LUTs per probe. Also, one BRAM per probe is needed since it is FIFO-based.

C. Accuracy of Performance Debugging

The in-FPGA flow is cycle accurate by its nature and does not need accuracy testing. The same applies to the size of DRAM transactions on simulation flow. For the evaluation of the cycle accuracy of the simulation flow, we can use the in-FPGA flow to compare the exact cycles of each module.

TABLE II: Logic overhead of monitors for in-FPGA flow.

# of probes	LUT	DSP	BRAM
4	654	0	4
16	2802	0	16
35	6072	0	35

In addition to the quicksort example we provided, we tested HLScope on matrix multiplication, convolutional neural network, and logistic regression. For each application, we classified the submodules into compute-intensive and DRAM-intensive and averaged the absolute difference. The result in Table III shows that the average error rate is 2.2% for compute modules and 7.0% for DRAM modules.

TABLE III: Cycle accuracy of the software simulation flow.

	Compute-intensive			DRAM-intensive		
Type	#mod	AVG($\|Dif\|$)	σ	#mod	AVG($\|Dif\|$)	σ
Quicksort	33	7.3%	0.050	5	8.4%	0.003
Conv NN	1	0.05%	-	3	1.8%	0.016
Mat mul	1	0.04%	-	3	12.4%	0.021
Log reg	3	1.4%	0.019	1	5.3%	-
AVG	-	2.2%	-	-	7.0%	-

V. Concluding Remarks

We have discussed how HLScope can help FPGA programmers easily identify performance bottlenecks. HLScope provides automated source-to-source transformation to easily extract the required parameters. We also have proposed a pure-HLS in-FPGA monitoring for accurate analysis. The experimental results show that the simulation flow requires 31% overhead in HLS's software simulation, but is orders of magnitude faster than bitstream generation. This flow is accurate within 2.2% on average for compute PEs. The in-FPGA flow consumes only about 170 LUTs and a BRAM per monitored module and provides cycle-accurate results.

Acknowledgment

We would like to thank Xilinx for the FPGA donation, and Falcon Computing and Xilinx for the software donation.

References

[1] Alpha Data, Alpha Data ADM-PCIE-7V3 Datasheet, 2013, *http://www.alpha-data.com/pdfs/adm-pcie-7v3.pdf*.

[2] Y. Choi, *et al.*, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," in *Proc. DAC*, 109–114, 2016.

[3] J. Curreri, *et al.*, "Performance analysis framework for high-level language applications in reconfigurable computing," *ACM Trans. Reconfigurable Technology and Systems*, 3(1), 2009.

[4] R. Deville, I. Troxel, and A. George, "Performance monitoring for run-time management of reconfigurable devices," in *Proc. IEEE Int. Conf. Engineering of Reconfigurable Systems and Algorithms*, 175–181, 2005.

[5] D. Finley, Optimized QuickSort, 2007, *http://alienryderflex.com/quicksort*.

[6] Intel, Intel FPGA SDK for OpenCL, 2016, *http://www.altera.com/*.

[7] Intel, Intel VTune Amplifier, 2017, *http://www.intel.com/*.

[8] S. Koehler, J. Curreri, and A. George, "Performance analysis challenges and framework for high-performance reconfigurable computing," *Parallel Computing*, 34(4–5):217–230, 2007.

[9] J. Lancaster, J. Buhler, and R. Chamberlain, "Efficient runtime performance monitoring of FPGA-based applications," in *Proc. IEEE Int. System-on-Chip Conf.*, 2009.

[10] P. Li, P. Zhang, L. Pouchet, and J. Cong, "Resource-aware throughput optimization for high-level synthesis," in *Proc. Int. Symp. FPGA*, 200–209, 2015.

[11] NVIDIA, NVIDIA Nsight, 2017, *http://www.nvidia.com/*.

[12] ROSE compiler infrastructure, 2017, *http://rosecompiler.org/*.

[13] A. Schmidt, N. Steiner, M. French, and R. Sass, "HwPMI: an extensible performance monitoring infrastructure for improving hardware design and productivity on FPGAs," *Int. J. Reconfigurable Computing*, 2012.

[14] Xilinx, SDAccel Development Environment, 2016, *http://www.xilinx.com/*.

[15] Xilinx, Vivado High-level Synthesis UG902, 2016, *http://www.xilinx.com/*.