HLScope+: Fast and Accurate Performance Estimation for FPGA HLS

Young-kyu Choi*†, Peng Zhang†, Peng Li‡, and Jason Cong*
*University of California, Los Angeles †Falcon Computing Solutions ‡Tsinghua University
{ykchoi,cong}@cs.ucla.edu, pengzhang@falcon-computing.com, lipeng@tsari.tsinghua.edu.cn

***Abstract*—High-level synthesis (HLS) tools have vastly increased the productivity of field-programmable gate array (FPGA) programmers with design automation and abstraction. However, the side effect is that many architectural details are hidden from the programmers. As a result, programmers who wish to improve the performance of their design often have difficulty identifying the performance bottleneck. It is true that current HLS tools provide some estimate of the performance with a fixed loop count, but they often fail to do so for programs with input-dependent execution behavior. Also, their external memory latency model does not accurately fit the actual bus-based shared memory architecture. This work describes a high-level cycle estimation methodology to solve these problems. To reduce the time overhead, we propose a cycle estimation process that is combined with the HLS software simulation. We also present an automatic code instrumentation technique that finds the reason for stall accurately in on-board execution. The experimental results show that our framework provides a cycle estimate with an average error rate of 1.1% and 5.0% for compute- and DRAM-bound modules, respectively, for ADM-PCIE-7V3 board. The proposed method is about two orders of magnitude faster than the FPGA bitstream generation.**

I. Introduction

High-level synthesis (HLS) tools such as Xilinx's Vivado HLS [27] and Intel's HLS [12] are being widely used to improve the productivity of field-programmable gate array (FPGA) programmers. HLS tools automatically transform a design written in high-level languages into a low-level implementation. By abstracting away the hardware execution model, HLS has shortened the learning curve of accelerator design. HLS also allows quick modification of various design choices such as pipeline initiation intervals and unrolling factors, therefore enabling programmers to perform efficient design space exploration.

However, one of the side effects of such automation and abstraction is that HLS hides many architectural details from the programmer. A programmer can only observe a brief synthesis report and a machine-generated output code, which is almost impossible to comprehend. As a result, if programmers wanted to analyze the output code for further performance improvement, they often have to spend many hours to identify the performance bottleneck.

Our previous work, HLScope [8], addresses this problem by providing an HLS-based framework that helps programmers to identify the performance bottleneck and its cause. The flow provides the execution time of each module and analyzes various stall causes, such as external DRAM access, synchronization, and dependency stalls. As will be reviewed in Section II, HLScope provides a source-to-source (S2S) transformation that automates the measurement and analysis process. HLScope provides two flows; a software simulation-based flow for rapid analysis and an in-FPGA flow for accurate analysis.

However, there are several issues that has not been adequately studied in HLScope. The first issue is the accuracy of the simulation-based flow when the program has dynamic execution paths and undetermined loop bounds that depend on the input data. A motivating example for a well-known quicksort is shown in Fig. 1 [10] (also used in [8]). Depending on the value of pivot, the number of iterations for Loop 1.1.1 and 1.1.2 can vary from 1 to N. Also, it may or may not execute some of the conditional statement (*e.g.* code A, B, and C). The synthesis report by Vivado HLS for this program is shown in Table I; it does not provide any information about the total execution time. HLScope partially solves this problem with simple variable trip count (TC) estimation method [8], but it is accurate only within 50% of the true cycle (Section III-C). This is due to the partly unavailable cycle information, which will be explained in Section II-B2.

```
void qsort_comp(int n_per_batch, float arr[MAX_N]){
  int beg[M], end[M], i=0, L, R; float piv, swap;
  beg[0]=0; end[0]=n_per_batch;
  while (i>=0) { // Loop 1            //one round of reordering
    L=beg[i]; R=end[i]-1;                    // init L & R ptr
    if (L<R) {
      piv=arr[L];
      while (L<R) { // Loop 1.1       // swap until L & R meets
        while (arr[R]>=piv && L<R){ // Loop 1.1.1
        #pragma HLS pipeline    // decrement R until an element
          R--;                  // less than pivot is found
        }
code A  if (L<R) { arr[L++]=arr[R]; }          // copy it to L
        while (arr[L]<=piv && L<R){ // Loop 1.1.2
          #pragma HLS pipeline  // increment L until an element
          L++;                  // less than pivot is found
        }
code A  if (L<R) { arr[R--]=arr[L]; }          // copy it to R
      }
      arr[L]=piv;beg[i+1]=L+1;end[i+1]=end[i];end[i++]=L;
      if (end[i]-beg[i]>end[i-1]-beg[i-1]) { //swap qsort order
code B  swap=beg[i]; beg[i]=beg[i-1]; beg[i-1]=swap;
        swap=end[i]; end[i]=end[i-1]; end[i-1]=swap;
    }}
    else {
      i--;  code C
} } }
```

Fig. 1: Code for non-recursive quicksort [10]

TABLE I: Vivado HLS report for non-recursive quicksort. IL, II, and TC are explained in Section II-B.

Name	IL	II	TC
qsort_comp	?	-	-
Loop 1	?	-	?
Loop 1.1	?	-	?
Loop 1.1.1	4	4	?
Loop 1.1.2	4	4	?

Another issue is the high-level external memory modeling for fast cycle estimation. Vivado HLS does allow setting a latency term for each port, but this is a too-optimistic prediction that does not consider the effective DRAM bandwidth and the memory access contention. For an example design that has one read and one write port of 512b external memory with very long burst access, Vivado HLS will provide a cycle estimate with the assumption that the kernel may have a DRAM bandwidth of 25.6 GB/s (=2*200MHz*512b). However, [7] reports that the effective bandwidth would be from 4.9 GB/s to 9.5 GB/s depending on the platform and the configuration.

The final issue lies in the limited applicability of the HLScope's in-FPGA monitoring when finding source of stalls. Both Intel OpenCL HLS (default) and Xilinx Vivado HLS (optional) support streaming dataflow communication where modules execute without any explicit synchronization among them. However, HLScope cannot provide a stall reason in this situation because the stall reason was statically analyzed (detailed reason will be explained in Section II-C).

Our work, named HLScope+, addresses these shortcomings by providing a fast and accurate HLS-based cycle estimate of the FPGA execution. In order to compensate for the inaccuracy introduced by the code segments with unknown cycle information, we develop a HLS-specific code instrumentation technique to extract the best estimate of these inaccuracy sources from the HLS synthesis report. This will be explained in Section III.

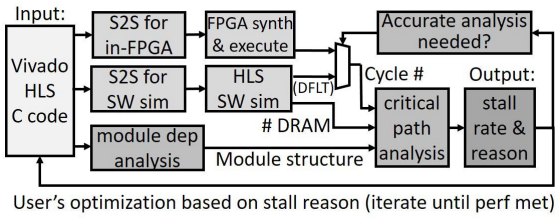

Fig. 2: Performance debugging overall framework [8]

We also provide a high-level external memory model for a typical FPGA architecture, where multiple processing elements (PEs) are connected through a bus to the shared external memory. We assume the PEs issue memory requests of various data bitwidth and burst length. For reduced estimation time, the external memory model is incorporated into the HLS software simulation. One of the challenge is how to abstract the individual memory access into a high-level model for fast estimation. Another challenge is that outstanding memory access from multiple PEs occurs in parallel, whereas the software simulation is performed in serial. The solution will be explained in Section IV.

Finally, we will describe how to build an in-FPGA monitor that infers the reasons for stall among dataflow modules connected through FIFO. A stall analysis network (SAN) is proposed that enables each module to trace the root cause of stall. The method is scalable since the monitor logic increases linearly with the number of modules under analysis. Also, the monitor is described purely in C code without the need for RTL, as will be demonstrated in Section V.

II. Review of HLScope

A. Overall Framework

In this section we review the basics of the HLS-based performance debugging framework named HLScope [8]. The flow is shown in Fig. 2. HLScope accepts a Vivado HLS C code for input. Next, it instruments a hardware execution model at HLS C code level for software simulation. The instrumentation is performed using APIs in the ROSE compiler infrastructure [22]. The software simulation is performed in Vivado HLS, and the instrumented code outputs the number of cycles and the DRAM transaction bytes for each module. With the module structure, cycle, and DRAM transaction information, the tool finds the list of the most time-consuming modules (performance critical path). Then the stall rate of each module is computed by dividing the individual module execution time by the performance critical path time. Also, the reason for stall is obtained by analyzing the module structure. Since the simulation-based estimation may have some inaccuracy, the performance critical path analysis may not be exact when modules running in parallel have similar cycle numbers. Then HLScope will recommend the actual on-board (in-FPGA) execution. Also, programmers may manually choose this flow if accurate analysis is the main concern. HLScope provides an automated instrumentation to insert in-FPGA monitors for this purpose.

B. Cycle Estimation with Software Simulation

1) Dependency Analysis: Rather than providing a cycle estimate for individual lines of operations, HLScope provides the analysis in blocks of code such as functions and loops. Suppose that a code block p is composed of multiple nested blocks $c = c1, c2, ...$ HLScope will add the cycle estimate for blocks c to block p depending on the static dependency analysis of the code. There are three possibilities: serial, parallel, and parallel by dataflow. If true dependency exists between blocks, they can be executed in a serial manner. The execution time of c (t_c) will be added to p (t_p) as:

$$t_p = \sum_c t_c \tag{1}$$

If there is no dependency, the blocks can execute in parallel. The execution time is the maximum of the parallel blocks:

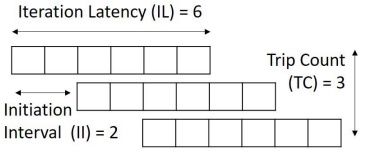

Fig. 3: Loop pipelining parameters

```
int loop_11_cycle = 0;
while (L<R) {  // Loop 1.1  // Swap until L ptr & R ptr meets
  ......
  int loop_111_cycle = 0;
  int loop_111_TC=0;
  while (arr[R]>=piv && L<R){ // Loop 1.1.1 //Decrement R ptr
    #pragma HLS pipeline      //until an element less than
    R--;                      //pivot is found or L & R meets
    loop_111_TC++;
  }
  loop_111_cycle = (loop_111_TC-1)*loop_111_II + loop_111_IL;
  loop_11_cycle += loop_111_cycle;
  ......                      // Swap arr[L] and arr[R]
}
```

Fig. 4: Code instrumentation to find dynamic loop bound for loop 1.1.1 in Fig. 1. Instrumented code is in bold. [8]

$$t_p = \max_c (t_c) \tag{2}$$

The final case is when there is a dependency between blocks, but the user would like them to execute in parallel without explicit synchronization. This requires that data be passed in a streaming fashion and the modules to be connected through FIFO. Also, keyword "*#pragma HLS dataflow*" should be used [27]. Since blocks are executed in parallel, the execution time is also bounded by the slowest block (Eq. 2). The difference is that they do have dependency between them, so it takes $\sum_c IL_c$ for data to traverse from beginning to end:

$$t_p = \max_c (t_c) + \sum_c IL_c \tag{3}$$

We will refer to this case as *parallel by dataflow*.

2) Loop Analysis: The parameters used to estimate the execution cycle of a pipelined loop can be explained with Fig. 3. The number of cycles to complete one iteration of a loop is called the iteration latency (IL). The execution of each iteration is pipelined, and the input rate of the pipeline is called the initiation interval (II). The number of iterations is referred to as trip count (TC). As can be deduced from Fig. 3, the execution cycle for a pipelined loop is [18]:

$$t = II * (TC - 1) + IL \tag{4}$$

However, for loops with undetermined loop bound (see quicksort example in Fig. 1), Vivado HLS cannot provide the cycle estimate due to the existence of undetermined TC. The tool will only provide the II and IL. We can obtain the actual TC by inserting a simple counter statement, as shown in bold statements in Fig. 4 [8]. Regardless of the existence of undetermined loop bound or conditional break statement, TC can be correctly estimated. Then the loop cycle estimation code is inserted after the loop based on the run-time acquired TC. Finally, the cycle estimate for Loop 1.1.1 is hierarchically added into its parent loop (Loop 1.1) cycle estimate.

However, the cycle estimate obtained from such technique is only accurate within 50% of the true cycle (Section III-C) for the quicksort example. The reason is that quicksort contains imperfect loops (loop 1 and 1.1) that cannot be obtained with variable TC estimation methods or initiation interval information from the HLS report. Also, quicksort contains dynamic execution paths (conditional statements in code A, B, and C) that are unavailable in the report. The solution will be discussed in Section III.

C. Cycle Extraction from In-FPGA Monitoring

For accurate analysis, HLScope provides a flow that inserts hardware monitor into the FPGA design and collects cycle information

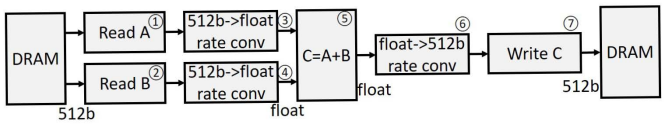

Fig. 5: Dataflow vector add connected through FIFO. Circled number is the module ID.

in runtime. Each module under analysis sends the start and the end of execution signal to the monitor. Then the monitor records this cycle information and writes to the DRAM for offline analysis. Similarly, work in [24] explains how to extract cycle and memory access information for designs written in OpenCL by recording event timestamps in trace buffers.

Although HLScope collects cycle information in runtime, the stall reason analysis is based on the statically analyzed module dependence. This is due to the difficulty in dynamically correlating the stall reason among multiple modules, especially when the origin is several hops away. As a result, HLScope cannot trace the exact source of stalls when modules execute in parallel by dataflow because the source of stall changes in runtime. For a simple example of vector add in Figure 5, all modules execute as soon as they receive a chunk of data through the streaming FIFO but will stall when the input data or the output buffer is not available. Module 5 (C=A+B), for example, will stall even if only one of the modules in the A/B read pipeline (FIFO being empty) or C write pipeline (FIFO being full) stalls. The solution will be discussed in Section V.

III. Improving Cycle Estimation Accuracy for Loops and Conditional Statements

A. Estimation for Loops

The instrumentation technique in Section II-B [8] allows the cycle estimate of an unbounded loop to be found, but an accurate estimate cannot be obtained if loops are not perfectly nested. An example can be found in Loop 1.1. Even if the cycles for Loop 1.1.1 and 1.1.2 are found by the TC instrumentation, there is no information about code A. The reason is that Vivado HLS did not give the IL of Loop 1.1 ('?' for Loop 1.1's IL in Table I) because of the existence of unbounded Loops 1.1.1 and 1.1.2. The same problem exists for Loop 1.

This problem can be formalized as follows. Suppose that a loop p is composed of multiple nested blocks $c = c1, c2, ...$, and one of the blocks, $c1$, is a loop. p is not a perfectly nested loop due to the existence of $c2, c3, ...$ Assuming true dependency exists among $c1, c2, ...$, the iteration latency of the p (IL_p) is the sum of c blocks' execution time (t_c):

$$IL_p = \sum_{c} t_c = \sum_{c \neq c1} t_c + II_{c1} * (TC_{c1} - 1) + IL_{c1}. \quad (5)$$

If the trip count of $c1$ (TC_{c1}) is not known at compile time, the HLS compiler will be unable to provide IL_p even if the rest of nested loops' execution time ($\sum_{c \neq c1} t_c$) is known. The variable trip count instrumentation (Fig. 4 [8]) on TC_{c1} allows estimation of $II_{c1} * (TC_{c1} - 1) + IL_{c1}$ in Eq. 5, but the estimate is not accurate since p's non-perfectly nested region ($\sum_{c \neq c1} t_c$) has not been considered.

To solve this problem, we can force the HLS compiler to provide the missing information $\sum_{c \neq c1} t_c$ by assigning an arbitrary TC_{c1}. In Vivado HLS, this can be achieved with the pragma "*#pragma HLS loop_tripcount min=100 avg=100 max=100*" where 100 is the arbitrary trip count. With this instrumentation, HLS will provide IL_p, II_{c1}, TC_{c1}, and IL_{c1}. Then the imperfect loop part is estimated by subtracting $(II_{c1} * (TC_{c1} - 1) + IL_{c1})$ from IL_p. Whether the arbitrary TC_{c1} matches the actual trip count in execution is irrelevant, since IL_p also increases at the exact same rate of II_{c1} (Eq. 5). If there are multiple nested loops (c1,..,cN) in p, the cycle estimation for the non-perfectly nested region of p can be generalized as:

$$\sum_{c \neq c1,..,cN} t_c = IL_p - \sum_{c=c1,..,cN} (II_c * (TC_c - 1) + IL_c). \quad (6)$$

```
while (i>=0) {
  #pragma HLS loop_tripcount min=100 avg=100 max=100
  ……
    while (L<R) {
      #pragma HLS loop_tripcount min=100 avg=100 max=100
      while (arr[R]>=piv && L<R){
        #pragma HLS loop_tripcount min=100 avg=100 max=100
        ……
      }
      while (arr[L]<=piv && L<R){
        #pragma HLS loop_tripcount min=100 avg=100 max=100
        ……
      }
    }
    ……
}
```
Fig. 6: Quicksort code after pragma insertion

TABLE II: HLS report for quicksort after code modification

Name	IL	II	TC
qsort_comp	401~8111401	-	-
Loop 1	4~81114	-	100
Loop 1.1	811	-	100
Loop 1.1.1	4	4	100
Loop 1.1.2	4	4	100

For the quicksort example, HLScope+ first automatically inserts the pragma on every loop to make Vivado HLS assume that the loop bounds are fixed (Fig. 6). This allows Vivado HLS to generate a report in Table II. An estimate for the non-perfectly nested region of Loop 1.1 can then be obtained by subtracting the cycle estimate of Loops 1.1.1 and 1.1.2 from Loop 1.1 (811-400-400 =11). The estimate for Loop 1 can be obtained in a similar way (81114-811*100 =14). The estimates for Loop 1.1 and Loop 1 will be automatically inserted into the simulation code as shown in Fig. 7. In the software simulation process, the HLS estimate from the arbitrary loop bounds will be ignored and will instead be estimated as Eq. 4 with the instrumented TC.

B. Estimation for Conditional Statements

The cycle estimate for most conditional statements will be automatically reflected because the cycle addition routine will only be processed when certain paths have been executed. However, some conditionals that exist between loops or functions will not be properly processed. For example, Vivado HLS does not provide a separate estimate for the if and the else part of the conditional statements in code A, B, and C.

We compensate for the missing cycle information based on the minimum cycle provided in HLS report. Suppose that Vivado HLS provides a range of cycle estimate for multiple execution paths in the kernel (*e.g.*, Loop 1 entry of Table II). If the simulation code for one of execution paths has a cycle estimate smaller than the minimum cycle provided by HLS tool, the cycle difference is added to the simulated code. Although exact cycle is still unknown, the cycle for unknown blocks are at least partially compensated.

For the quicksort example, Loop 1 has a minimum latency of 4. The execution path along the else part of code C has no known latency, but a cycle estimate of (4-0) is added as shown in Fig. 7.

Since this is a minimum analysis, the cycle estimate may have some error. In practice, however, most of time-consuming blocks are in the form of loops, and Vivado HLS will report II of those loops. Thus, we can still obtain an accurate estimate with the proposed techniques, as will be evaluated in Section VI-B.

C. Estimation Result for Quicksort

After applying the two techniques explained in this section, our tool automatically generates the code presented in Fig. 7. Running software simulation on this instrumented code with N=131072 elements provides a prediction of 14.6M cycles, which is a 2.0% difference from the cycle-accurate RTL co-simulation (Table III). The estimation

```
int qsort_cycle = 0; // global variable
qsort_cycle += (8111401-81114*100);
……
int loop_1_cycle = 0;
while (i>=0) {  // loop 1
  ……
  if (L<R) {
    loop1_cycle += (81114-811*100);
    piv=arr[L];
    int loop_11_cycle = 0;
    while (L<R) { // loop 1.1
      loop_11_cycle += (811-400-400);
      int loop_111_tc=0;
      while (arr[R]>=piv && L<R){ // loop 1.1.1
        ……
        loop_111_tc++;
      }
      loop_11_cycle += (loop_111_tc-1)*4+4;
      ……
          <loop 1.1.2 is similar to loop 1.1.1>
      ……
    }
    loop_1_cycle += loop_11_cycle;
    ……
    if (end[i]-beg[i]>end[i-1]-beg[i-1]) {
      ……
  } }
  else {
    loop1_cycle += (4-0);
    ……
} }
qsort_cycle += loop1_cycle;
```

Fig. 7: Quicksort code after inserting cycle estimation code

time is four orders of magnitude faster. Compared to the baseline software simulation, the instrumented code incurred a small 2.5% time overhead. Also, the final instrumented version provides a 48% more accurate cycle estimate than the simple TC counting version.

TABLE III: Cycle estimation and simulation time for quicksort

	Baseline	Simple TC counting[8]	+ imperfect loop & cond stmt est	RTL co-sim
Cycle Est	N/A	7.42M (-50%)	14.6M (-2.0%)	14.9M (0%)
Sim Time	0.0394s (1X)	0.0395s (1.002X)	0.0404s (1.025X)	817 (20736X)

IV. External Memory Access Model for HLS Software Simulation

For our external memory access model, we assume an environment where there are several PEs on an FPGA, and they are connected to a single external memory through a bus and a DRAM controller. We do not model cache and instead assume that FPGA programmers explicitly control the internal BRAM as a scratchpad memory for higher performance. We also assume that the bus can accept several outstanding requests from PEs, as is the case with the AXI4 bus standard [25].

In Vivado HLS, the simulator will assume that data can be fetched from the external memory every cycle. As a result, the estimated DRAM BW will be ($\#ext\_port * bitw * f_{bus}$), where $\#ext\_port$ is the number of external memory port, $bitw$ is the data bitwidth, and f_{bus} by the frequency of the bus. As mentioned in the introduction, it will assume an ideal DRAM bandwidth of 25.6 GB/s for a design with $\#ext\_port$=2, $bitw$=512b, and f_{bus}=200MHz, but only 4.9GB/s~9.5GB/s is achieved during on-board testing [7].

A simple estimation model for a single DRAM transaction is [19]:

$$t_{MEM} = DSIZE/DBW + DLAT \quad (7)$$

where the $DSIZE$ is the length of each transaction, DBW is the DRAM bandwidth, and $DLAT$ is the latency. $DSIZE$ is obtained from the size field of the memcpy function. If the access is in the form of global array reference, it is found from the length of the consecutive array index from the loop bound of the loop iterator. Vivado HLS also informs programmers if it has flattened several loops to extend the consecutive access length. Similar to the cycle estimate routine, the length is used as a variable that is determined in simulation in run time. Note that if $DSIZE$ is larger than the maximum bus burst length (1KB for AXI), Vivado HLS will automatically divide it into several outstanding memory requests of maximum bus burst length.

We test two boards: ADM-PCIE-7V3 [1] and ADM-PCIE-KU3 [2]. In ADM-PCIE-7V3, DBW has been reported as 9.5 GB/s for read and 8.9 GB/s for write in [7], but this number cannot be achieved when the access is not consecutive or if the bus bitwidth varies. The refined model will be explained in the following subsection. The latency term has been measured as $DLAT_R$ =542ns and $DLAT_W$ =356ns. In ADM-PCIE-KU3, we have obtained DBW_R =10.3 GB/s, DBW_W =9.6 GB/s, $DLAT_R$ =434ns, and $DLAT_R$ =325ns.

A. Bandwidth Model Refinement

We refine the DRAM bandwidth (DBW) model by assuming it is bounded by all components in the memory access pipeline: physical DRAM module bandwidth (DBW_P), DRAM controller bandwidth (DBW_C), and the bus data bandwidth (DBW_B). The effective bandwidth is computed as:

$$DBW = min(DBW_P, DBW_C, DBW_B) \quad (8)$$

1) Physical DRAM Bandwidth: In ADM-PCIE-7V3, we found that when short (4B) but many (>1MB) discrete memory data are accessed, the effective external memory bandwidth is reduced to about 7% (0.053GB/s) of the bandwidth achieved with the consecutive memory access of the same length (0.75GB/s). The reason can be found in the bandwidth constraint of the physical DRAM module. For illustration, let us consider consecutive outstanding requests to random memory location: *for(int i = 0; i < N; i++){ bram[i] = dram[bram_rand_addr[i]];}*.

Assuming a closed row policy [16], the minimum time between access in a different address in a DRAM module is $t_{RC} = t_{RAS} + t_{RP}$, where t_{RC} is the row cycle time, t_{RAS} is the row address select (RAS) time, and t_{RP} is the RAS precharge time [11]. For the ADM-PCIE-7V3 and ADM-PCIE-KU3, the DRAM specification [14] states that $t_{RAS} = 36ns$, $t_{RP} = 13.5ns$, and $t_{RC} = 49.5ns$. In practical implementation, this theoretical latency is often exceeded, and extra overhead is added on the controller side (t_{CO})—that is, the latency becomes t_{RC} + t_{CO}. From the random access experiment, we found that the t_{RC} + t_{CO} is 76ns, which suggests that the controller overhead (t_{CO}) is (76ns-49.5ns)=26.5ns in ADM-PCIE-7V3. In ADM-PCIE-KU3, t_{CO} is calculated as (62ns-49.5ns)=12.5ns. Note that the average latency of 76ns for 4B of data is 0.053GB/s (=4B/76ns), which is the bandwidth obtained in the random access experiment.

We found that discrete memory access of stride 2, 4, and 8 achieve similar external memory bandwidth as the random access. This suggests that the Xilinx controller does not concatenate outstanding requests of nearby memory addresses into a same burst DRAM access. Thus, we model strided access in the same way as the random access.

If requested data length is larger than the number of DRAM data pins $\#dq = 128b$, each $\#dq$ bits of data will be sent every $f_{ddr} = 1.33GHz$ cycle. In addition to t_{RC} + t_{CO}, the initial overhead includes RAS to column address select (CAS) time (t_{RCD}) added to CAS time (t_{CAS}), which is 13.5ns. The CAS signal is given in parallel to RAS signal. In summary, the physical DRAM module access time (t_{D_P}) and DBW_P can be approximated as:

$$t_{D_P} = max(t_{RAS}, t_{RCD} + t_{CAS} + len/f_{ddr}) + t_{RP} + t_{CO} \quad (9)$$

$$DBW_P = len * \#dq/t_{D_P} \quad (10)$$

where len is the burst length of requested data in unit of $\#dq$ bits.

2) Bus Data Bandwidth: If the kernel's external port data bitwidth ($bitw$) is less than the maximum bus data bitwidth supported (512b for SDAccel), the overall DRAM bandwidth might be limited by the bus data bandwidth (DBW_B). DBW_B is computed by multiplying $bitw$ (*e.g.,* float: 32b, uint512: 512b) by the frequency of the bus(f_{bus}):

$$DBW_B = f_{bus} * bitw; \tag{11}$$

3) DRAM Controller Bandwidth: Since we do not have knowledge of the inner workings of Xilinx's DRAM controller propriety, it is difficult to construct a good model for the DRAM controller bandwidth (DBW_C). Thus, we indirectly measure it by putting many outstanding long consecutive access requests (>512MB) with maximum bus data size (512b).[1] Since DBW_P and DBW_B achieve their peak values in this test method, the measured bandwidth, if lower than DBW_P and DBW_B, can be assumed to be DBW_C. As mentioned previously, the bandwidth for ADM-PCIE-7V3 is measured as 9.5GB/s for read and 8.9GB/s for write, which is smaller than $DBW_{Pmax} = 21GB/s$ and $DBW_{Bmax} = 12GB/s$. Thus, DBW_C is set to 9.5GB/s for read and 8.9GB/s for write.

B. Modeling Multiple PE Contention

In this section we explain the cycle estimation process when multiple PEs try to access the memory at the same time. Rather than using a time-consuming cycle-accurate transaction model that accounts for individual DRAM access, we propose a high-level estimation method for our fast software simulation-based framework. As an example, we consider the for loop (p) in Fig. 8 which contains three PEs ($c = c1, c2, c3$): *load()*, *qsort_comp()*, *store()*.

Since we assume a single external memory controller, contention among PEs may incur additional delay. Suppose that a block p contains $c = c1, c2, ...$ PEs, and PE $c1$ (*e.g.*, *load()*) has m_{c1} external memory transfers. According to Eq. 7, the execution time of $c1$ (t_{c1}) would consist of memory transfer time ($t_{MEM_{c1}} = \sum_{m_{c1}}(DSIZE_{m_{c1}}/DBW_{m_{c1}} + DLAT)$) and computation time ($t_{COMP_{c1}}$). However, t_{c1} may become larger than ($t_{COMP_{c1}} + t_{MEM_{c1}}$) if the data transfer time of other PEs c (*e.g.*, *store()*) executing in parallel cannot be completely overlapped with the non-data-transfer time of PE $c1$, that is, ($t_{COMP_{c1}} + \sum_{m_{c1}} DLAT$). This is expressed in Eq. 12:

$$t_{c1} = \sum_{m_{c1}} \frac{DSIZE_{m_{c1}}}{DBW_{m_{c1}}} + \max(t_{COMP_{c1}} + \sum_{m_{c1}} DLAT, \sum_{c \neq c1} \sum_{m_c} \frac{DSIZE_{m_c}}{DBW_{m_c}}) \tag{12}$$

However, a major challenge in implementing Eq. 12 is that the software simulation process is usually sequential, and we cannot model the concurrent execution of memory access. For the example in Fig. 8, *load()*, *qsort_comp()*, and *store()* will execute in parallel in hardware, but the software simulation process will compute them serially. Then, the estimation routine in *load()* does not have the knowledge of $DSIZE_{store}$ or DBW_{store}, since *store()* has not been executed yet. Even if the HLS software simulation process does allow multi-threaded execution in the future so that *store()* and *load()* can communicate in flight, having a mutex or semaphore for hundreds of PEs will significantly slow down the simulation and diminish the advantage of having a high-level prediction.

We solve this problem by carrying additional estimation results without $DLAT$ and resolving the summation part of Eq. 12 hierarchically. For each PE $c1$, we accumulate only the data term ($DSIZE/DBW$) of t_{MEM}, excluding the latency term ($DLAT$):

$$t_{BO_{c1}} = \sum_{m_{c1}} DSIZE_{m_{c1}}/DBW_{m_{c1}} \tag{13}$$

[1]Note that this measurement method itself coincides with the method used in [7], but [7] used this method to obtain DRAM bandwidth (DBW), whereas we use it to obtain one constraint (DBW_C).

```
for( int i = 0 ; i < N ; i++ ){
  load(dram_portA, bram_load[0], ...);
  qsort_comp(bram_load[1], bram_store[1], ...);
  store(dram_portB, bram_store[0], ...);
}
```

Fig. 8: Code example for external memory access from multiple PEs, where *load()*, *qstore_comp()*, *store()* have no dependency with double buffering

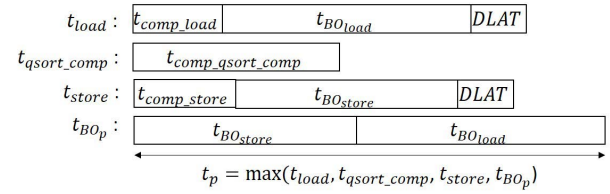

Fig. 9: Computing cycle estimate (Eq. 15 and Eq. 16) for the example given in Fig. 8, in the for loop that contains $load()$, $qsort\_comp()$, $store()$ PEs

where $t_{BO_{c1}}$ is the minimum bus occupation cycle of PE $c1$. The insight behind this term is that the computation part and the DRAM latency might be overlapped in multiple outstanding requests, but at least $t_{BO_{c1}}$ is needed for each data in PE $c1$ to be transferred through the bus.

We also keep track of t_{c1}, as if there was only one PE:

$$t_{c1} = t_{COMP_{c1}} + t_{MEM_{c1}} \tag{14}$$

Next, we can hierarchically compute t_{BO_p} and t_p of the for loop:

$$t_{BO_p} = \sum_c t_{BO_c} \tag{15}$$

$$t_p = \max(\max_c(t_c), t_{BO_p}) \tag{16}$$

Eq. 15 represents that the bus occupation cycle of p is the sum of c's bus occupation cycles t_{BO_c}. This implies that the access to external memory is serialized. Finally, Eq. 16 [2] represents that the overall execution time is the maximum of c's individual latencies and p's minimum bus occupancy cycle. Unlike Eq. 2 which simply takes the maximum of nested PEs' cycles, the proposed equation can account for the case where *combined* DRAM access cycles of multiple PEs dominate the compute time. This can be more easily understood graphically in Fig. 9, where t_{BO_p} is the largest term in Eq. 16 because the system is memory bound.

Since the proposed estimation method does not account for the exact ordering of memory access, the estimate of the individual parallel-running module's execution time is not guaranteed to be exact. For example, in Fig. 9 it is uncertain when load() or store() submodule will exactly terminate. However, what this model does predict is the *combined* execution time of parallel-running modules. This is more important since the performance critical path and the stall reason is evaluated based on the most time-consuming module (Section II-A). The accuracy evaluation will be presented in Section VI-B.

V. In-FPGA Stall Analyzer for FIFO-based Dataflow Application

This section describes an in-FPGA HLS-based stall reason monitoring instrumentation method for modules executed in parallel by dataflow (Section II-B) that communicate through FIFOs and have no explicit synchronization. As explained in the vector add example (Fig. 5), it is difficult to find the root cause of stall, because any module that is stalled will cause all other modules in the read or write pipeline to stall when the FIFO becomes empty or full. We solve this problem by instrumenting a series of probes into modules

[2]It is also possible to verify that inserting Eqs. 13, 14, and 15 into Eq. 16 will give the same equation if Eq. 12 is inserted into Eq. 2. Also, note that the max function in Eq. 16 can be replaced with an add function if the submodules are executed in serial rather than in parallel.

```
void read_A(..., hls::stream<uint512> &fifo_512_r,
           hls::stream<bool> &p_status, &p_active, ... )
{
  uint512 bram[1024];
  for (int i = 0; i < vec_size ; i++) {
    memcpy( bram, global + i * 1024, 1024*64 );
    p_status.write(1); //start of inter-mod comm pipeline
    for (int j = 0; j < 1024; j++) {
#pragma HLS PIPELINE II=1
      p_active.write(1); //pipeline active
      fifo_512_r.write(bram[j]);
    }
    p_status.write(0); //end of inter-mod comm pipeline
  }
  p_exit_mtr.write(0); p_exit_fifo.write(0); //module end
}
```

Fig. 10: Code instrumentation for *Read A* module. Instrumented code in bold.

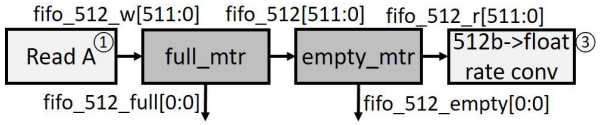

Fig. 11: Instrumentation of *full_mtr* and *empty_mtr* for FIFO

under analysis (MUA) and FIFOs. Also, we instantiate a network of the probe processing logic (*monitors*). The details will be presented in the following subsections.

A. Code Instrumentation for Module Under Analysis

We start by regaining the visibility of the module status (in execution or stalled) that is hidden due to HLS abstraction. Fig. 10 describes the process for module 1 (*Read A*), and Table IV lists the series of probes used. Probe *p_status* is inserted to indicate whether the module is in inter-module communication mode or not. *p_status.write(1)* is inserted before start of the pipeline that contains the inter-module communication (*fifo_512.write()*), and *p_status.write(0)* is inserted to signal the end of pipeline. It is also possible to assign an extra status bit to further differentiate DRAM access mode.

TABLE IV: List of probes for module under analysis

Probe name	Description
p_status	Is in inter-module communication mode?
p_active	Is the communication pipeline not stalled?
p_exit_mtr	Terminate signal for stall analysis monitor.
p_exit_fifo	Terminate signal for instrumented FIFO.

Even if the module is executing the inter-module communication pipeline, it may stall if the FIFO is full or empty. To observe this, we instrument *p_active* that writes '1' if the communication pipeline is active. If stalled, it will not write anything.

Finally, we also instrument logic termination signal (value of 0) for the monitor logic (*p_exit_mtr*) and instrumented FIFOs (*p_exit_fifo*) that will be explained in the following subsections.

B. Code Instrumentation for FIFOs

The activeness of the module under analysis can be observed using the previous techniques, but if multiple FIFOs are connected to a module, it is difficult to find which FIFO is causing the stall. For this analysis, we also instrument measurement logics into the inter-module FIFOs. An example is given for the FIFO between module 1 and module 3 in Fig. 5. Two measurement logics are inserted between the write and the read module: *full_mtr* and *empty_mtr* that generate the fullness (*fifo_512_full*) and emptiness (*fifo_512_empty*) signal of the FIFO (Fig. 11).

The detailed code of the *full_mtr* logic is shown in Fig. 12. If fifo write to *fifo_512* is blocked, it will send '1' via wire *fifo_A_full* to the monitor logic (explained in the next subsection) to signal fullness of the FIFO. Also, all data reads and writes are performed in non-blocking mode, and the pipeline is set to II of 1. As a result, the

```
void full_mtr(hls::stream<uint512> & fifo_512_w, fifo_512,
           hls::stream<bool> & fifo_512_full, p_exit_fifo)
{
  bool loop_exit = 0;
  float data_buffer; // temporary data buffer
  bool data_in_buffer = 0; //data exists in data_buffer?

  while( loop_exit == 0 || data_in_buffer == 1 ){
#pragma HLS pipeline II=1
    if( data_in_buffer == 0 ){ // data_buffer empty
      if( fifo_512_w.read_nb(data_buffer) == 1 ){
        if( fifo_512.write_nb(data_buffer) == 0 ){
          fifo_512_full.write(1); // FIFO is full
          data_in_buffer = 1; //data stored to data_buffer
    } } }
    else{ // data_buffer not empty
      if( fifo_512.write_nb(data_buffer) == 1 ){
        if( fifo_512_w.read_nb(data_buffer) == 0 ){
          data_in_buffer = 0; // data_buffer empty
      } }
      else{
        fifo_512_full.write(1); // FIFO is full
    } }
    bool exit_data;
    if( p_exit_fifo.read_nb(exit_data) == 1 ){
      if( exit_data == 0 ){
        loop_exit = 1; // terminate monitoring
    } }
} }
```

Fig. 12: Code for *full_mtr* logic

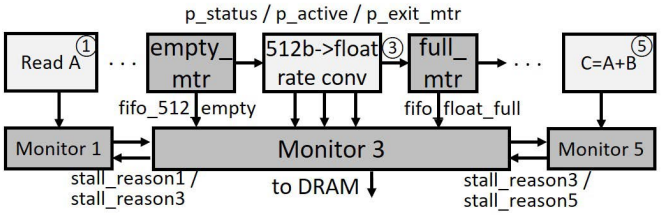

Fig. 13: Distributed stall analysis network

throughput of the original FIFO is maintained, and the instrumented logic does not cause additional stall to the original logic. In addition, a temporary data buffer is used to store a read data that was not written due to output being stalled. Though omitted, similar code is used for *empty_mtr* logic that monitors the emptiness of the FIFOs.

C. Monitor for Stall Analysis Network

In order to analyze the stall reason for module 5, for example, we would have to observe the status of all read pipelines (modules 1, 2, 3, 4) and write pipelines (modules 6, 7). However, this is not scalable since an interconnect of quadratic complexity would be needed between all MUAs and all monitors.

Instead, we propose a distributed stall analysis network (SAN). Part of the proposed logic is shown in Fig. 13. One monitor is instantiated per module under analysis. Each monitor classifies its host MUA as being active, or being stalled due to its neighbor. As described in Table VI, module being "active" is the period when the module is not in inter-communication mode (*p_status=0*) or when it is in communication and the pipeline is active (*p_status=1* and *p_active=1*). The module being stalled is when the module is in communication and the pipeline is not active (*p_status=1* and *p_active=N/A*). The reason for stall can be found by observing the empty or full signal sent from the monitored FIFO logic. If multiple stall reasons are asserted, we designed the monitor to arbitrarily select one of the reasons for the simplicity of the implementation.

If the FIFO port that causes the stall is identified, each monitor will record the stall reason sent from a neighbor monitor as being the root cause of the stall. This reason is generated at each monitor. If the MUA is active (*p_status=0 || (p_status=1 && p_active=1)*),

TABLE VI: List of states in the SAN monitor 3

Monitor state		Active		Stalled due to 1	Stalled due to 5
Input	p_status	0	1	1	1
	p_active	-	1	N/A	N/A
	f_512_empty	-	-	1	0
	f_float_full	-	-	0	1
Output	st_reas3	3		st_reas1	st_reas5
Register	act_cyc	+=1		-	-
	st_cyc[st_reas1]	-		+=1	-
	st_cyc[st_reas5]	-		-	+=1

the monitor will broadcast its own ID to its neighbor monitors. If its MUA is inactive (*p_status=1 and p_active=N/A*), it will pass on the stall reason (*st_reasX*) of the FIFOs that have been stalled.

We record the amount of cycles for each monitor state (active or stalled due to particular module) in registers (*act_cyc* or *st_cyc[X]*). When monitoring has finished (*p_exit_mtr=0*), each monitor will write its register contents to the DRAM for offline analysis.

Since each module passes the stall reason from its neighbors, it can identify the stall module even if it is several hops away. Also, SAN is scalable because the connection of each monitor is limited to its MUA, instrumented FIFOs, and the neighbor monitors.

VI. Experimental Results

A. Experimental Setup

For our evaluation platform, we use Alpha Data's ADM-PCIE-7V3 board [1] that has Xilinx's Virtex 7 690T FPGA and two Kingston's DDR3L-1333 SDRAMs[14]. We also use the ADM-PCIE-KU3 board [2] that has Xilinx's Ultrascale KU060 FPGA and two Kingston's DDR3L-1333 SDRAMs. For the FPGA synthesis, we use Xilinx's SDAccel 2016.2 [26] and Vivado HLS 2016.3 [27] software tools.

The benchmark we used includes quicksort [10], Cholesky [20], convolutional neural network [28], matrix multiplication [13], logistic regression [3], decompression [17], and compression [9]. We also use four applications from MachSuite [21], which is a collection of common applications for accelerator environments. We exclude some applications that were similar to other benchmarks or had some functional correctness problem. The original code has been optimized with pipelining, double buffering, data reuse, and duplication.

B. Performance Estimation Accuracy

To evaluate the accuracy of our software simulation-based estimation model, we use the in-FPGA cycle extraction flow [8] so that the exact cycle count of individual submodules can be obtained. For FIFO-based dataflow modules, we compare the number of active cycles obtained from SAN monitors with the estimated result. The applications are classified as having blocks with explicit synchronization (modules are only in serial or parallel) or parallel by dataflow (Section II-B). Also, we classified the submodules inside each application as being mainly compute-bound or DRAM-bound.

The estimation error is obtained by averaging the absolute difference between on-board testing and the simulated results. The comparison between 'Base' and 'Opt' column of Table V shows that the average error rate has reduced from 6.6% to 1.1% for compute-bound modules and 41% to 13.6% for DRAM-bound modules on ADM-PCIE-7V3 after applying the proposed methods in Section III and Section IV. For ADM-PCIE-KU3, the averaged error rate after optimization is 2.5% for compute-bound and 22% for DRAM-bound on the same benchmarks.

The accuracy improvement using techniques introduced in Section III was most effective for applications that had imperfect loops and conditional statements with known minimum latency, such as the compute modules in quicksort (55% $\rightarrow$ 4.0%) and Cholesky (21% $\rightarrow$ 0.56%). The bandwidth model proposed in Section IV reduces the estimation error rate of kernels that utilize only a part of the bus data bitwidth (*e.g.*, 256b port in CNN and 64b ports in SPMV). Also, the multiple PE contention model (Section IV-B) produces more accurate result for kernels with multiple DRAM-accessing modules (*e.g.*, 3 DRAM modules in vector add: 33% $\rightarrow$ 2.2%).

Even with the proposed improvements, some inaccuracy still exists. The reason for inaccuracy in the compute part is the missing cycle information for some of the dynamic execution paths (Section III-B). The inaccuracy in DRAM part is due to constructing a high-level behavioral model (Section IV), including the multiple PE contention model, rather than simulating each memory access. In fact, the error rate is relatively higher for designs with more DRAM-bound modules executed in parallel. However, as mentioned in Section IV-B, our model is more focused on making accurate predictions for modules on the performance critical path, so that the performance debugging framework can correctly analyze the stall reason. The error rate of the submodule with the longest execution time among parallel-executing submodules is shown in 'Opt-Long' column of Table V. As expected, the error rate has decreased from 13.6% to 5.0% for ADM-PCIE-7V3. For ADM-PCIE-KU3, it decreases from 22% to 9.4%. This suggests that the proposed modeling is reliable for performance debugging.

C. Software Simulation Flow Overhead

The software estimation overhead consists of two parts: first, the code instrumentation for hardware cycle estimation and second, overhead in the software simulation process. The code instrumentation takes 5-98 seconds. The software simulation overhead depends on the computational complexity of the original code compared to the inserted code. The comparison between the original software simulation time and the instrumented code software simulation time is shown in Table VII. It shows that overhead is 4% on average. The code instrumentation and simulation overhead is approximately two orders of magnitude faster than the FPGA bitstream generation. This shows that the proposed flow is suitable for rapid analysis.

TABLE V: Cycle estimation error of the software simulation flow on ADM-PCIE-7V3. (Base=Baseline estimation with variable TC estimation method (Section II-B2) and simple DRAM modeling (Eq. 7); Opt=Optimized with proposed techniques in Section III and Section IV; Opt-Long=Most time-consuming DRAM-bound submodule among parallel-executing submodules in Opt version)

Dep Type	Appl Name	Compute-bound			DRAM-bound			
		#subm	AVG($\|Dif\|$) Base	AVG($\|Dif\|$) Opt	#subm	AVG($\|Dif\|$) Base	AVG($\|Dif\|$) Opt	AVG($\|Dif\|$) Opt-Long
Explct Synch	Qsort[10]	33	55%	4.0%	5	88%	8.5%	8.5%
	Cholesky[20]	1	21%	0.56%	2	77%	0.57%	0.17%
	ConvNN[28]	1	0.05%	0.05%	3	36%	1.5%	0.31%
	Mat mul[13]	1	0.04%	0.04%	3	10%	37%	12%
	Log reg[3]	3	1.4%	1.4%	1	23%	0.76%	0.76%
	AES[21]	1	3.2%	3.2%	2	42%	34%	8.0%
	KMP[21]	1	0.36%	0.36%	1	25%	0.90%	0.90%
	NW[21]	1	0.03%	0.02%	2	75%	43%	14%
	SpMV[21]	1	3.24%	3.24%	2	76%	12%	12%
Paral by data flow	Vecadd	4	0.0%	0.0%	3	33%	2.2%	2.7%
	Mat mul[13]	6	0.05%	0.05%	6	33%	30%	2.0%
	Decomp[17]	3	1.3%	0.92%	2	8.2%	0.17%	0.32%
	Compr[9]	3	0.23%	0.23%	3	2.4%	4.9%	3.6%
AVG		-	6.6%	1.1%	-	41%	13.6%	5.0%

TABLE VII: Time overhead of SW simulation flow. Consists of code instrumentation and additional SW simulation time (unit:s).

Appl Name	Instr Time	SW Sim Unmodif	Instr SW Sim Est	Bitstr Gen
Qsort[10]	27	0.026	0.029 (1.12X)	1h27m
Cholesky[20]	5	0.083	0.089 (1.07X)	36m
ConvNN[28]	64	60	64 (1.07X)	1h47m
Mat mul[13]	43	62	65 (1.05X)	2h23m
Log reg[3]	36	563	564 (1.0X)	1h34m
AES[21]	51	62	65 (1.05X)	3h29m
KMP[21]	9	128	129 (1.01X)	1h21m
NW[21]	8	56	59 (1.05X)	1h38m
SpMV[21]	12	7.3	7.4 (1.01X)	2h5m
Vecadd	37	0.20	0.21 (1.05X)	1h30m
Mat mul[13]	76	125	125 (1.0X)	4h12m
Decomp[17]	98	0.80	0.80 (1.0X)	1h28m
Compr[9]	91	19	20 (1.05X)	5h35m
AVG		(1.0X)	(1.04X)	

D. Logic Overhead for SAN

SAN requires extra logic for on-board implementation. Each probe (*e.g., p_active* and *p_status*) in a MUA consumes about 20 LUTs. The instrumented FIFO consumes LUT that approximately increases linearly with the data bitwidth, as shown in Table VIII. The monitor resource usage slightly increases with the number of neighboring monitors, as shown in Table IX. None of them consume any DSP or BRAM.

TABLE VIII: Logic overhead of instrumented FIFO

	empty_mtr		full_mtr	
bitw	float	uint512	float	uint512
LUT	46	526	49	529

TABLE IX: Logic overhead of SAN monitor

# of neighbors	1	2	3
LUT	903	930	957

VII. Related Work

Other previous work on rapid performance estimation includes Aladdin [23], which constructs a dependence graph directly from the C code and produces a fast cycle estimate before going to RTL construction. Lin-analyzer [29] considers various FPGA-specific resources (*e.g.,* DSP, BRAM) during scheduling. The work in [15] uses a machine-learning based statistical model to predict the performance. These works focus on providing analysis for efficient exploration of possible design points. Our work is more targeted toward evaluating an actual design that is synthesized by an HLS tool.

Aladdin [23] and Lin-analyzer [29] can also be used to provide cycle estimate for programs with dynamic behavior since they utilize the instruction trace generated in C simulation. However, collecting instruction trace takes a relatively long time compared to our high-level cycle estimator (based on native C software simulation). For example, instruction trace generation for sorting 131,072 elements in Aladdin took 188 seconds, whereas the software simulation in our framework took 0.0404 seconds (Table III).

LegUp HLS [6] provides a flow to obtain hardware cycle estimation by profiling the software for the number of times each basic block is executed. Then it multiplies the obtained execution number by the basic block cycle given by Legup HLS. Our work, on the other hand, is more focused on how to instrument the code to extract unknown basic block cycles that are apparently hidden by HLS. Thus, HLScope+ only requires high-level synthesis reports like Table II and does not need to extract HLS LLVM compiler's internal data, which could be proprietary information.

For external memory modeling, [23] links a cycle-accurate trace-based DRAM simulator to a performance estimator, but cycle-accurate simulation takes a long time compared to high-level modeling. Work in [4], [5] describes approximately timed transactional-level SystemC simulation with instruction set simulator (ISS), but we would like to raise the level of abstraction to a level of function call or loops, which speeds up the simulation process compared to modeling individual external memory requests. A simple high-level model for a single external memory accessor has been provided in [19], but FPGA typically has multiple processing elements competing for the shared external memory resource. The work in [15] uses a statistical model to estimate the external memory latency of fetching a block, but it is uncertain that such predefined template models for block access can be generalized for arbitrary access patterns.

VIII. Concluding Remarks

We have described a high-level cycle estimation methodology for input-dependent FPGA designs using the HLS software simulation process. A source-to-source code instrumentation technique was used to automate the cycle extraction process. Also, we provided a high-level estimation model for DRAM access for a typical bus-based architecture with outstanding requests and multiple PEs. In addition, a stall analysis network was proposed for cycle-accurate in-FPGA analysis of FIFO-based dataflow applications. Experiments on an ADM-PCIE-7V3 board showed that our estimation has an error rate of 1.1% in compute-bound modules and 5.0% in performance-critical DRAM-bound modules, with a modest 4% time overhead in software simulation.

IX. Acknowledgment

This work is in part supported by NSF InTrans Award with additional supports from Fujitsu and Intel. We would like to thank Xilinx for the FPGA and the software donation. We also thank Peng Wei for optimizing Machsuite benchmarks and Janice Wheeler for proofreading this manuscript.

References

[1] Alpha Data, Alpha Data ADM-PCIE-7V3 Datasheet, 2017, *http://www.alpha-data.com/pdfs/adm-pcie-7v3.pdf*.
[2] Alpha Data, Alpha Data ADM-PCIE-KU3 Datasheet, 2017, *http://www.alpha-data.com/pdfs/adm-pcie-ku3.pdf*.
[3] Apache Spark examples, *http://spark.apache.org/examples.html*.
[4] L. Benini, *et al.*, "SystemC cosimulation and emulation of multiprocessor SoC designs," *Computer*, 53–59, 2003.
[5] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proc. Int. Conf. Hardware/software Codesign and System Synthesis*, 19–24, 2003.
[6] A. Canis, *et al.*, "From software to accelerators with LegUp high-level synthesis," in *Proc. Int. Conf. CASES*, 18–26, 2013.
[7] Y. Choi, *et al.*, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," in *Proc. DAC*, 109–114, 2016.
[8] Y. Choi and J. Cong, "HLScope: High-Level performance debugging for FPGA designs," in *Proc. Int. Symp. FCCM*, 2017.
[9] J. Cong, *et al.*, "CPU-FPGA co-optimization for big data applications: A case study of in-memory Samtool sorting," in *Proc. Int. Symp. FPGA*, 291, 2017.
[10] D. Finley, Optimized QuickSort, 2007, *http://alienryderflex.com/quicksort*.
[11] IBM, Application Note: Understanding DRAM Operation, 1996.
[12] Intel, Intel FPGA SDK for OpenCL, 2016, *http://www.altera.com/*.
[13] J. Jang, S. Choi, and V. Prasanna, "Energy-and time-efficient matrix multiplication on FPGAs," *IEEE T. VLSI*, 13(11):1305–19, 2005.
[14] Kingston, KVR13LSE9/8 memory module specifications, 2012, *http://www.kingston.com/datasheets/*.
[15] D. Koeplinger, *et al.*, "Automatic generation of efficient accelerators for reconfigurable hardware," in *Proc. ISCA*, 2016.
[16] C. Lee, O. Mutlu, V. Narasiman, and Y. Patt, "Prefetch-aware DRAM controllers," in *Proc. Int. Symp. Microarchitecture*, 200–209, 2008.
[17] J. Lei, *et al.*, "A high-throughput architecture for lossless decompression on FPGA designed using HLS," in *Proc. Int. Symp. FPGA*, 277, 2016.
[18] 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.
[19] J. Park, P. Diniz, and K. Shayee, "Performance and area modeling of complete FPGA designs in the presence of loop transformations," *IEEE T. Computers*, 53(11):1420–1435, 2004.
[20] L. Pouchet, PolyBench/C, 2015, *http://web.cse.ohio-state.edu/pouchet.2/software/polybench/*.
[21] B. Reagon, *et al.*, "Machsuite: Benchmarks for accelerator design and customized architectures," in *Proc. IISWC*, 110–119, 2014.
[22] ROSE compiler infrastructure, *http://rosecompiler.org/*.
[23] Y. Shao, *et al.*, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proc. ISCA*, 97–108, 2014.
[24] A. Verma, *et al.*, "Developing dynamic profiling and debugging support in OpenCL for FPGAs," in *Proc. DAC*, 56–61, 2017.
[25] Xilinx, AXI Reference Guide UG761, 2012, *http://www.xilinx.com/*.
[26] Xilinx, SDAccel Development Environment, 2016, *http://www.xilinx.com/*.
[27] Xilinx, Vivado High-level Synthesis UG902, 2016, *http://www.xilinx.com/*.
[28] C. Zhang, *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. Int. Symp. FPGA*, 161–170, 2015.
[29] G. Zhong, *et al.*, "Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators," in *Proc. DAC*, 136–141, 2016.