Rapid Cycle-Accurate Simulator for High-Level Synthesis

Yuze Chi, Young-kyu Choi,* Jason Cong, and Jie Wang
Computer Science Department, University of California, Los Angeles
{chiyuze,ykchoi,cong,jiewang}@cs.ucla.edu

ABSTRACT

A large semantic gap between the high-level synthesis (HLS) design and the low-level (on-board or RTL) simulation environment often creates a barrier for those who are not FPGA experts. Moreover, such low-level simulation takes a long time to complete. Software-based HLS simulators can help bridge this gap and accelerate the simulation process; however, we found that the current FPGA HLS commercial software simulators sometimes produce incorrect results. In order to solve this correctness issue while maintaining the high speed of a software-based simulator, this paper proposes a new HLS simulation flow named FLASH. The main idea behind the proposed flow is to extract the scheduling information from the HLS tool and automatically construct an equivalent cycle-accurate simulation model while preserving C semantics. Experimental results show that FLASH runs three orders of magnitude faster than the RTL simulation.

ACM Reference Format:
Yuze Chi, Young-kyu Choi, Jason Cong, and Jie Wang. 2019. Rapid Cycle-Accurate Simulator for High-Level Synthesis. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3289602.3293918

1 INTRODUCTION

Although FPGA has many promising features including power-efficiency and reconfigurability, the low-level programming environment makes it difficult for programmers to use the platform. In order to solve this problem, many high-level synthesis (HLS) tools such as Xilinx Vivado HLS [9] and Intel OpenCL HLS [14] have been released. These tools allow programmers to design FPGA applications with high-level languages such as C or OpenCL. This trend is reinforced by recent efforts on FPGA programming with languages of higher abstraction—such as Spark or Halide [21, 25].

Even though such progress has been made on the design automation side, a large semantic gap still exists on the simulation side. Programmers often need to use low-level register-transfer level (RTL) simulators and try to map the result back to HLS. The result is often incomprehensible to those who are not FPGA experts. Moreover, such low-level simulation takes a very long time. Some work has been done to automate hardware probe insertion from the HLS source file [4, 12, 18, 22]; however, this work requires regeneration of FPGA bitstream if there is a change in the debugging point, and the turnaround time is often in hours.

These problems can be partially solved by the software-based simulators provided by HLS tools. It takes little time to reconfigure the debugging points, and no semantic gap exists between the simulation and the design. However, a well-known shortcoming of these simulators is that most of them do not provide performance estimation. In addition, we found a critical deficiency—they sometimes provide *incorrect* results.

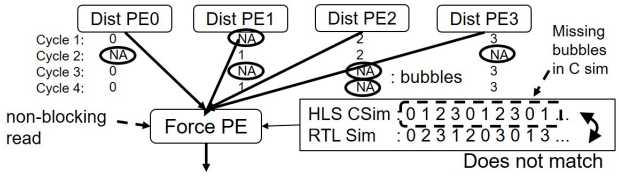


Figure 1: Molecular dynamics simulation PEs [7]

An example can be found in the molecular dynamics simulation [7] (Fig. 1). Multiple distance processing elements (Dist PEs) filter out faraway molecules above threshold and send them to Force PE. The pruned molecules will create a bubble (empty data) in the FIFO, and Force PE will process only the valid data (after non-blocking read) in the order they are received from any of the FIFOs. However, if the modules are instantiated in the order of (Dist PE1, PE2, ... Force PE) in the source file, Vivado HLS will finish the simulation of Dist PE1 first, followed by Dist PE2, and so on. As a result, by the time Force PE is simulated, the bubbles in the FIFOs are completely removed, and the Force PE output ordering can be entirely different from the actual result. If one was analyzing the DRAM access behavior from the HLS simulation output, the person would likely draw a wrong conclusion.

Another problematic example can be found in the artificial deadlock situation [11], which occurs when the depth of the FIFO is smaller than the latency difference among modules (details in Section 3.2). The first issue is that the HLS software simulator cannot detect the deadlock situation and proceeds as if there is no problem with the design. The second issue is that after we apply a transformation to remove the deadlock, the HLS tool cannot also simulate the amount of performance degradation (Section 7.3) from the artificial stall (Section 3.2). We also found a problem in the simulation of feedback loops where the feedback data is ignored by the HLS tool (Section 3.3).

The primary reason for the incorrect simulation result is that HLS software simulators do not guarantee cycle accuracy. The comparison between the software simulator of the two most popular ([17]) commercial FPGA HLS tools, Xilinx Vivado HLS and Intel OpenCL HLS, is presented in Table 1. Vivado HLS assumes unlimited FIFO depth which makes it difficult to accurately model FIFO fullness/emptiness. Also, their sequential simulation execution model prevents correctly simulating designs with feedback

Table 1: Comparison of the software-based simulation of Xilinx Vivado HLS [24] and Intel OpenCL HLS [14]. Undesirable characteristics are in bold.

	Xilinx Viv HLS C Sim	Intel OpenCL HLS Sim
FIFO depth	**Unlimited**	Exact
Exec model	**Sequential**	Concurrent
Feedback	**Not supported**	Supported
Sim speed	~5 Mcycle/s	**~1 Mcycle/s**
Sim order	Deterministic	**Non-deterministic**
Cycle-acc	**Not cycle-accurate**	**Not cycle-accurate**

*Corresponding author.

FPGA '19, February 24–26, 2019, Seaside, CA, USA

ACM ISBN 978-1-4503-6137-8/19/02…$15.00
https://doi.org/10.1145/3289602.3293918

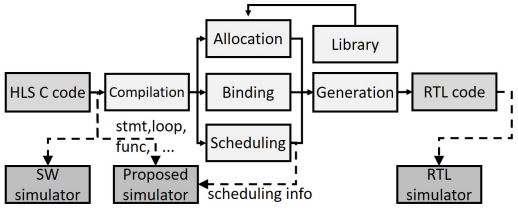


Figure 2: HLS design steps [10] and simulation flows

loops (Section 3.3). Intel OpenCL HLS simulates about 5X slower than Vivado HLS, but it correctly simulates the FIFO depth. The tool assigns a thread to each module for concurrent simulation; however, the execution order of the threads is not deterministic and may produce different results in different simulation runs for cases in Section 3.

HLS design steps and conventional simulation flows are shown in Fig. 2. A software simulator runs fast but provides no cycle estimation and may have the correctness problem. An RTL simulator is accurate but runs slow since it incorporates low-level implementation details. Our solution to these problems is based on the idea that it may be possible to tackle both problems by simulating based on the scheduling information. It would be faster than the RTL simulation without the allocation / binding information and the component libraries; and it would solve the correctness problem of the software simulation and provide accurate performance estimation with its cycle-accuracy.

Although simulating solely based on the scheduler output (LLVM IR + scheduling information) is a possible option, we have instead decided to simulate in C syntax and augment it with scheduling information. The reason is that we wanted to raise the simulation abstraction level to further accelerate the simulation process and also make it easier for programmers to understand what is being simulated. To our knowledge, this is the first HLS-based simulation flow that takes such an approach.

By taking such an approach, however, several challenges were encountered (will be elaborated in Section 4). One problem is how to model high-level semantics such as functions and loops—as well as FIFO transactions and FIFO stalls—in a cycle-accurate fashion. Moreover, correctly simulating the task-level and pipelined parallelism that is inherent in hardware (and the corresponding RTL simulation) in sequential C semantics is a significant challenge.

In this paper we propose FLASH[1][2]—an HLS-based software simulation flow that addresses these challenges. We describe transformations that allow cycle-accurate simulation of communication and computation stages (will be explained in Section 4). Also, a method will be explained to simulate multiple levels of parallelism with C semantics. These steps will be described in Section 5.

We obtain the scheduling information from the HLS synthesis report and automatically generate a new simulation code based on the information. The new simulation code was made compatible with the conventional HLS software simulator for easy integration with the existing tool. The overall flow is described in Section 6.

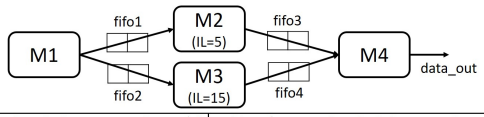
Our current initial version is based on Vivado HLS, but we hope to extend our work to Intel HLS if the tool provides detailed internal scheduling information in the future.

2 RELATED WORK

Work in [4, 12, 18, 22] describe frameworks that allow users to specify debugging points in high-level language and synthesize hardware probes into the FPGA for analysis. They can be categorized into work that has more focus on verifying functional correctness [12, 18] and work that has more focus on extracting performance-related parameters [4, 22]. Compared to the software-based flows, however, these hardware-based debuggers typically requires hours of initial overhead for bitstream generation.

There are several SystemC simulators [6, 20] that can achieve cycle-accuracy for the source code that has explicit scheduling information specified by the programmer, but this may be too difficult for non-experts. Our flow, on the other hand, achieves cycle-accuracy for a HLS C source code that does not have such user-defined scheduling information.

There are also other HLS-based software simulators. The LegUp HLS [2] simulator provides speedup prediction based on the profiling of the source code and the execution cycle from its synthesis result. HLScope+ [5] describes a method to extract cycle information that is hidden by HLS abstraction and uses Vivado HLS C simulation to predict the performance for applications with dynamic behavior. These works, however, do not guarantee cycle-accuracy.




```
void M1(stream<int>& f_out1, .. f_out2){
  for (int i = 0; i < N/16; k++) {
    for (int j = 0; j < 16; k++) {
#pragma HLS pipeline II=1
      f_out1.write(16*i+j);
      f_out2.write(16*i+j+10);
} } }

void M2(stream<int> & f_in, ... f_out){
  for(int i = 0; i < N; i++){
#pragma HLS pipeline II=1
    int temp = f_in.read();
    f_out.write(temp*711);
} }

// M4 is omitted
```

```
void M3(stream<int>& f_in, ... f_out){
 for(int i = 0; i < N; i++){
#pragma HLS pipeline II=1
    int temp = f_in.read();
    f_out.write((int)((float)temp*3));
} }

void toy_mpath(int data_out[N]){
  stream<int> fifo1,fifo2,fifo3,fifo4;
#pragma HLS stream var=fifo1, .. depth=2
#pragma HLS dataflow
  M1(fifo1, fifo2);
  M2(fifo1, fifo3);
  M3(fifo2, fifo4);
  M4(fifo3, fifo4, data_out);
}
```

Figure 3: Structure and code for motivating example `toy_mpath`

3 PROBLEM DESCRIPTION AND MOTIVATING EXAMPLES

In this section we describe three classes of problems that cause current HLS tools to produce incorrect software simulation result. The problems are demonstrated with motivating examples in the literature.

3.1 Incorrect Data Ordering with Multiple Paths

Suppose a PE is reading data in a non-blocking fashion from multiple PEs through FIFOs as in the molecular dynamics simulation example (Fig. 1 [7]) in the introduction. If a bubble exists in a FIFO, the data consumer PE will skip the FIFO and proceed to read from the next FIFO. In software simulation, however, if the data producer PEs are instantiated in the source file before the consumer PE, Vivado HLS will simulate the data producer PEs completely before moving on the next one. This effectively removes all bubbles in the FIFO, and the order of output from the data consumers in the software simulation result will be different from the actual execution. In the Intel HLS, the simulation order of the data producers is undetermined, and thus there is no guarantee that the bubbles in the simulated result will exactly match the actual execution.

3.2 Artificial Deadlock and Stall

Consider an example in Figure 3 where the module `M2` has a latency of 5 and `M3` has a latency of 15. All FIFOs have a depth of 2. After `M2` has produced two output elements, `M4` cannot consume any of them because `fifo4` is still empty due to the long latency of `M3`. Due to the back-pressure from `M2` and `fifo3`, `fifo1` becomes full. Then

[1] FLASH: Fast, ParalleL, and Accurate Simulator for HLS

[2] An extended version of this paper is available at: https://arxiv.org/abs/1812.07012

```
for(i = 0; i < N; i++){
#pragma HLS pipeline II=1
  int temp = f_in.read();
  f_out.write(...temp*3);
}
```

```
01 i = 0;
02 while (i < N){
03 #pragma HLS pipeline II=1
04   if(f_in.empty()==false){
05     int temp = f_in.read();
06     f_out.write(...temp*3);
07     i++;
08 } }
```

Figure 4: Source-to-source code transformation to avoid artificial deadlock for M3 in Fig. 3

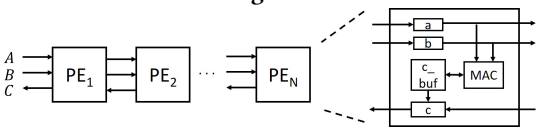


Figure 5: Matrix multiplication with linear systolic array architecture

M1 will stop producing output to fifo2 because fifo1 and fifo2 have to be written in the same cycle. fifo2 will eventually become empty, which blocks the pipeline of M3. Then none of the modules can do any further useful work, and the circuit deadlocks. This is called an artificial deadlock [11]. The deadlock is caused by the mismatching latency of multiple paths and the small FIFO depth. This can be observed in real applications, such as the dataflow-based architecture for stencil computations in [3] that contains various modules and FIFOs with different latencies and depths.

The problem is that software-based HLS simulators ignores the latency of a module. It will simulate each iteration of a loop as if the data is instantaneously passed from input to output. Thus Vivado HLS will proceed with the simulation as if the deadlock has not happened. Intel HLS compiler avoids the deadlock problem by automatically increasing the FIFO depth; however, this creates a new problem of mismatch between what is simulated and synthesized.

The second problem was found after we applied code transformation to avoid the deadlock. Figure 4 shows the transformation for M3 in Figure 3. If the input FIFO is empty, a bubble is inserted into the pipeline (line 4)—this allows the pipeline to keep processing the already-read data even if there is no additional input. The deadlock situation is removed since M4 can now receive the output from M3.

Even though the deadlock was avoided, however, the modules still have to wait for the data to be flushed. This causes a delay that we call *artificial stall*. Since HLS tools do not consider the delay due to the latency of a module, such performance degradation cannot be simulated.

3.3 Missing Data from Feedback Path

As mentioned previously, Vivado HLS simulates the functions in the order they are instantiated in the source code. This causes a problem if a feedback path exists that passes data from later instantiated functions to earlier ones. At the time earlier functions are simulated, the data would not be available. As a result, Vivado HLS simulates the program as if the feedback FIFOs are always empty. Intel HLS can simulate the feedback data from blocking read correctly, because a thread simulating each module can wait for others to pass the data—although it is not guaranteed that the feedback data from non-blocking read will arrive at the right timing.

We demonstrate this problem with matrix multiplication example ($C = A \times B$) in linear systolic array architecture [8, 16]. As shown in Fig. 5, each PE computes one column of the matrix C ($C_{ij} \mathrel{+}= A_{ik} * B_{kj}$). Data from the matrix A and B are fed into the array in the forward direction, while the results of matrix C are collected in the backward direction. If the modules are instantiated in the order of PE_1, PE_2, ..., and PE_N, Vivado HLS will simulate PE_1 assuming the FIFO for C is always empty, and this will cause the tool to produce incorrect results.

4 PROBLEM STATEMENT AND CHALLENGES

The data ordering problem (Section 3.1) can be solved if the simulator models the FIFO data transaction (read/write) and the FIFO stall (empty/full) in a cycle-accurate fashion. The artificial deadlock problem (Section 3.2) requires modules to initiate FIFO read and write at the timing that reflects the computation latency. In other words, it requires cycle-accurate modeling of *computation stages*, which we define as the computation latency between pairs of FIFO read and FIFO write. The feedback problem (Section 3.3) does not occur if the FIFO read in the feedback path is simulated after the FIFO write.

Thus, the problem is stated as follows: given a source code and its scheduling information, we need a simulator that models the communication and the computation stages in a cycle-accurate manner. The simulator also must produce correct output data.

In addition to this main requirement, the simulator should be able to provide the execution cycles of each module to help programmers apply performance optimization. Also, if the modules deadlock, the simulator should provide the content of the internal registers for debugging purpose. Moreover, the simulation code should be semantically similar to the source code as much as possible (as opposed to being a low-level code such as RTL), so that users can easily understand what is being simulated.

With such complicated requirements, several challenges arise:

- **Challenge 1 : Cycle-accurate simulation**
  It is difficult to discover the exact cycle when statements are executed since the information given by the HLS tool is very limited. Intel OpenCL HLS only provides loop initiation interval (II). Vivado HLS provides slightly more information—such as the module's finite-state machine (FSM) state when FIFO read or write is performed. However, for computation statements, it is difficult to find the exact cycle, because Vivado HLS only provides lists of LLVM IR and the corresponding FSM states. Mapping such low-level representation back to the original C code is a difficult task.
  Also, even if the schedule of all operations are known, the simulator has to *selectively* execute statements that correspond to a particular FSM state at each cycle. Moreover, the content of the variables in the previous state has to be available, and the updated variables have to be stored for the next state simulation.
- **Challenge 2 : Simulation of parallelism**
  RTL is an inherently parallel language—it has multiple levels of parallelism including task-level parallelism and pipelined parallelism. On the other hand, pure C is written in a sequential form. The challenge is in transforming C into a form that can simulate the concurrency.
- **Challenge 3 : FIFO communication and pipeline stall**
  In RTL simulation, a full or empty signal from FIFO can halt an FSM. An equivalent software simulator would also need to mimic this behavior based on the status of the FIFOs. Also, a deadlock would need to be detected if all pipelines can no longer make any progress.
- **Challenge 4 : Loop and function simulation**
  We would need to construct an equivalent model of high-level semantics, such as loops and functions.

5 AUTOMATED CODE GENERATION FOR RAPID CYCLE-ACCURATE SIMULATION

In this section, we provide a solution to each challenge in Section 4 and describe our proposed automated simulation code generation flow. For illustration, we will use the toy_mpath example (Fig. 3)

```
01 void M2_SIM(){                    //simulation function for M2
02   static int M2_state = 1;//use "static" var for the next cycle
03   ...
04   if(M2_state == 1){      //state conditional block for state 1
05     ...          //computation stmt & communication for state 1
06   }
07   else if(M2_state == 2){ //state conditional block for state 2
08     ...          //computation stmt & communication for state 2
09   }
10 }                 //exit sim function after simulating one cycle
```

Figure 6: Simulation function structure for cycle-accurate simulation

after applying the deadlock avoidance transformation discussed in Section 3.2.

5.1 Cycle-Accurate Simulation

For cycle-accurate simulation, we declare an FSM state variable for each module and copy statements to the conditional block that correspond to its simulated state. An example can be found for M2 module in lines 4–9 of Fig. 6. Only the statements for a single cycle are simulated and then the simulation function exits. The contents of the variables are restored and saved regardless of simulation function entrance or exit by using `static` variables (line 2).

Regardless of the exact cycle a computation statement is simulated, we exploit the fact that the behavior observed from outside the module (including the module's computation stage) would be the same as long as the inter-module FIFO communication is simulated at the correct cycle. Thus, even if the schedule of a module's computation statement is unknown, we can assign an arbitrary state that does not violate the timing causality with the cycle-known FIFO communication that has dependency with the computation statement. We assign states to the computation statements based on as-soon-as-possible scheduling policy to reduce the number of pipelined shift registers (Section 5.2.1). The simulation of computation statements and FIFO communication will be further explained in Section 5.2.1 and Section 5.3, respectively.

5.2 Simulation of Parallelism

5.2.1 Pipelined Parallelism. In a pipelined loop, different iterations are executed in parallel in a single FSM state. The parallel factor is same as the loop iteration latency (IL, also called pipeline depth). To simulate such parallelism, we need to keep multiple copies of the same variable for each pipelined stage. For example, the "temp" variable in M2 (Fig. 3) is copied through the pipeline like shift registers (line 15 of Fig. 7). Then, instead of placing the computation for each pipeline stage in a corresponding M2_state conditional block as in Fig. 6, we place all computation in a single M2_state conditional block as shown in lines 4–23 of Fig. 7. This transformation allows us to effectively simulate the pipelined parallelism. If II is larger than 1, the computation at state i is placed at the state conditional block of $i\%II$.

It is important to note that the order of each pipeline stage has been *reversed* (st6, ... st3, st2). This limits the content of shift register to be copied to the immediate next state only in a single cycle. Also, in order to invalidate a pipeline bubble (from the artificial deadlock avoidance transformation in Section 3.2), we propagate the enable signal through the pipeline stages (line 14 and 19).

5.2.2 Task-Level Parallelism. The task-level parallelism is simulated by processing one cycle of all modules and FIFOs in a round-robin fashion. This is processed in the scheduler loop in line 6-14 of Fig. 8. It is composed of module (line 8-9) and FIFO (line 10-11) simulation loop.

It is possible that different order of the module and FIFO simulation loop leads to different output—for example, depending on if

```
01 static bool p1_en_st3, ... p1_en_st6  = false; //enable signals
02 static int temp_st3, ... temp_st6;           //shift registers
03 ...
04 else if(M2_state == 2){ //starting state for the pipelined loop
05   ...
06   if( p1_en_st6 == true ){   //enabled 4 cycles after FIFO read
07     p1_en_st6 = false;       //disables enable signal after use
08     fifo3_arr[fifo3_wptr++] = temp_st6*711;  //FIFO data write
09     fifo3_wnum--;                            //(see Sect 5.3.1)
10   }
11   ...
12   if( p1_en_st3 == true ){    //enabled 1 cycle after FIFO read
13     p1_en_st3 = false;       //disables enable signal after use
14     p1_en_st4 = true;              //enable signal propagation
15     temp_st4 = temp_st3;   //shift register for "temp" variable
16   }
17   if( i_st2 < N ){         //loop exit condition (see Sect 5.4)
18     if( fifo1_rnum != 0 ){ //if FIFO not empty (see Sect 5.3.1)
19       p1_en_st3 = true; //enables if path for later pipe stages
20       temp_st3 = fifo1_arr[fifo1_rptr++];    //FIFO data read
21       fifo1_rnum--;                          //(see Sect 5.3.1)
22       i_st2++; ...       // loop iterator update (see Sect 5.4)
23 } } }
```

Figure 7: Code transformation to model cycle-accurate, pipelined parallelism (M2 in Fig. 3)

```
01 void (*MList[M])();                  /module func ptr list
02 void (*FList[F])();                   //FIFO func ptr list
03 Mlist[0] = M1_SIM;          .... Mlist[3] = M4_SIM;   //init
04 Flist[0] = fifo1;           .... Flist[3] = fifo4;
05
06 while(1){  //scheduler loop:
07   ...   // loop until until deadlock or all modules finish
08   for(i = 0; i < M; i++)            //simulate all modules
09     Mlist[i]();
10   for(i = 0; i < F; i++)              //simulate all FIFOs
11     Flist[i]();
12   ...
13   cycle++;
14 }
```

Figure 8: Module/FIFO simulation scheduler to model task-level parallelism

the data producer PE is simulated before or after the consumer PE. A way to avoid this problem will be discussed in Section 5.3.1.

5.3 FIFO Simulation

5.3.1 FIFO Communication. The FIFO is implemented as a circular buffer with read/write pointers (`fifo_rptr` and `fifo_wptr`) and an array (`fifo_arr`) of FIFO buffer size. Also, we declare `fifo_rnum` and `fifo_wnum` variables to denote the number of data and buffer space available in the FIFO. FIFO reads and writes in the source code are transformed based on Table 2. For example, the FIFO write in M2 (fifth line of M2 in Fig. 3) would be transformed to: ($fifo3\_arr[fifo3\_wptr\ ++] = temp\_st6 * 711$; $fifo3\_wnum\ --$;) (line 8-9 of Fig. 7).

In addition to decreasing the number of buffer space ($fifo3\_wnum\ --$;) for FIFO write, we would need to increase the number of available data ($fifo3\_rnum\ ++$;). However, this process is delayed until the FIFO simulation loop (line 10-11 of Fig. 8). The reason is to ensure that simulating data producer PE earlier than the consumer PE (in the module simulation loop in line 8-9 of Fig. 8) does not allow transfer of data through the FIFO in the same cycle (1 cycle latency is needed). More details on the FIFO simulation can be found in the extended paper (see footnote[2] on page 2).

Table 2: Code transformation for FIFO communication

HLS source code	Transformed simulation code
fifo.empty()	fifo_rnum == 0
fifo.full()	fifo_wnum == 0
data = fifo.read()	data = fifo_arr[fifo_rptr++]; fifo_rnum−−;
fifo.write(data)	fifo_arr[fifo_wptr++] = data; fifo_wnum−−;

```
j++;                  //inner loop update
if( !(j<16) ){          //inner loop cond
  i++;                //outer loop update
  j=0;                  //inner loop init
  if( !(i<N/16) ){    //outer loop cond
    M1_state = 0;           //loop exit
} }
```

Figure 9: Loop condition and update for flattened loop in M1 of Fig. 3

5.3.2 Pipeline Stall Modeling. If a pipeline stall condition is met, none of the statements should be simulated at the current state. Thus, the stall condition should be placed at the beginning of a state conditional block. This will make the simulation function to exit without changing any variables. After applying the artificial deadlock avoidance transformation, FIFO read no longer causes the stall, but FIFO write will. The stall condition is met when the FIFO is full and when the state for the FIFO write statement has been enabled. For example, the pipeline stall condition that corresponds to FIFO write in line 8 of Fig. 7 would be: $if(p1\_en\_st6$ && $fifo3\_wnum == 0)$. This condition will be added to line 5 of Fig. 7.

Note that our tool can detect a deadlock by checking if no state transition occurs (stalled) in any modules and no data transaction occurs in any FIFOs. This may happen if the user decides not to incorporate the artificial deadlock avoidance method (Section 3.2).

5.4 Loop and Function Simulation

Simulation of statements inside a pipelined loop has been discussed in Section 5.2.1. For the loop initialization statement, it is simulated upon entering the first state of a loop. The loop update expression is simulated at each iteration of a loop. If the loop condition is met after the update, state transition for loop exit occurs. For a flattened loop (e.g., M1 in Fig. 3), the update and the loop condition check is performed starting from the innermost nested loop, as illustrated in Fig. 9.

A function call is simulated by sending a module enable signal to the scheduler loop (Fig. 8). Next, the function argument values are copied into the newly called module.

6 OVERALL FLOW

The overall simulation framework of FLASH is shown in Fig. 10. Given an input Vivado HLS C code, we apply an optional preprocessing step of transforming pipelined loops to avoid artificial deadlock (Section 3.2). Also, some labels are added to easily identify loops and functions. The transformation step uses the APIs in the ROSE compiler infrastructure [19]. The transformed code is fed into the Vivado HLS for synthesis. Based on the scheduling report given by the HLS tool, the input code is automatically transformed for rapid software simulation (Section 5). The simulation code has been made compatible with the Vivado HLS software simulator for easy integration with the existing tool. As a final output, our flow currently provides the number of cycles consumed in each module. As a future effort it will be enhanced to provide both functional debugging support (e.g., data dump, triggers), and performance debugging support (e.g., module stall analysis).

7 EXPERIMENTAL RESULTS

7.1 Experimental Setup

For HLS tool, we use Vivado HLS 2018.2 [24]. For platform, we target the ADM-PCIE-KU3 board [1] with Xilinx's Ultrascale KU060 FPGA [23]. The target clock frequency is 250MHz. The simulation is conducted with a server node that has Intel Xeon Processor E5-2680 [13] and 64GB of DRAM. The simulation files were compiled with –O3 flag.

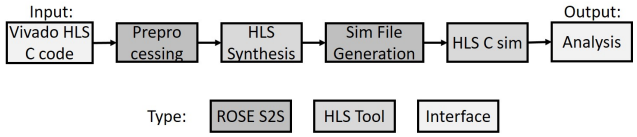


Figure 10: Overall simulation framework of FLASH

The experiment is performed on toy_mpath (Fig. 3) and three dataflow benchmarks: stencil [3], molecular dynamics simulation [7] (Fig. 1), and matrix multiplication [8] (Fig. 5).

7.2 Execution Time

As mentioned in Section 6, preprocessing, HLS synthesis, and simulation file generation steps are needed to prepare the files for the proposed simulation. The time breakdown of the steps is presented in Table 3.

Table 3: Simulation preparation time breakdown (preprocessing, HLS synthesis, and simulation file generation: Fig. 10)

Benchmark	Preproc	HLS Synth	SimFile Gen	Total
Toy_mpath	7.1s	24s	7.5s	39s
Stencil	15s	60s	22s	97s
MD_sim	8.0s	35s	11s	54s
Mat_mul	8.1s	31s	10s	49s

The simulation time comparison among Vivado HLS C simulation, Vivado HLS RTL simulation, Intel OpenCL HLS simulation (using Quartus 18.0 [15]), and our FLASH simulation flow is presented in Table 4. FLASH is about 1,390X (=1,570/1.13) faster than the RTL simulation. This confirms our initial speculation that simulating based on the scheduling information will result in much faster speed, since the simulation is not slowed by the resource allocation / binding information or the component library that exist in RTL simulation.

Table 4: Simulation time comparison among Vivado HLS C simulation, Vivado HLS RTL simulation, Intel OpenCL HLS simulation, and FLASH simulation

Benchmark	V C Sim	V RTL Sim	I OCL Sim	FLASH
Toy_mpath	0.602s (1.00X)	492s (817X)	4.60s (7.64X)	0.570s (0.947X)
Stencil	1.46s (1.00X)	113s (77.4X)	2.63s (1.80X)	1.25s (0.856X)
MD_sim	0.0547s (1.00X)	100s (1,830X)	0.0921s (1.68X)	0.0677s (1.24X)
Mat_mul	0.0539s (1.00X)	192s (3,560X)	0.201s (3.73X)	0.0810s (1.50X)
AVG	(1.00X)	(1,570X)	(3.71X)	(1.13X)

Since our flow reflects the scheduling information, we can expect some slowdown compared to the Vivado HLS C simulation. This is noticeable in Mat_mul, where the frequent FIFO stall (Table 5) lengthens the simulation process. MD_sim has a long simulation time due to the deep pipeline (55)—the overhead of copying shift registers and enable signals (Section 5.2.1) for pipeline stages becomes relatively large. However, it is interesting to note that for Toy_mpath and Stencil, FLASH was even faster than the Vivado HLS C simulation. This suggests that there was an unexpected factor which has negated the simulation speed overhead of the proposed flow. We found that this is largely attributed to the fact that Vivado HLS can allocate unlimited FIFO buffer for C simulation (Table 1). To model FIFO, the Vivado HLS C simulator uses the C++ Standard Template Library (queue.h), which incurs the overhead of

dynamically allocating buffer and copying its content. For example, the C simulation time of `Toy_mpath` reduces from 0.602s to 0.076s if we replace FIFO library calls with fixed-size arrays (array size is set to the number of total FIFO elements written). FLASH simulation flow does not have this problem, because the FIFO library calls have been replaced with array-based communication (Section 5.3). The average slowdown of FLASH compared to the Vivado HLS C simulation is 1.13X.

Please note that in our initial research stage, we also evaluated a similar flow with SystemC. However, the overhead in SystemC simulation environment was causing a 2-3X slowdown compared to the proposed C-based flow, which motivated us to follow the current approach.

7.3 Accuracy

As explained in Section 4, the correctness problem can be solved by simulating in a cycle-accurate manner. The data value and the data ordering has been verified by comparing the output of FLASH simulator with that of the RTL simulator.

In Table 5, we compare the cycle estimation accuracy with Vivado HLS synthesis report after we specify the maximum loop bound for each loop. We were not able to provide comparison with Intel HLS since the tool does not provide cycle estimate. The estimation error rate is small for `Stencil`, because [3] has built-in mechanism to allocate adequate buffers. For the rest of the benchmarks, we have applied a small (1–2) FIFO depth (an example was shown in Fig. 3). This causes FIFO buffer to be frequently full and empty and leads to worse performance than what HLS tool has predicted. Our flow, on the other hand, simulates in a cycle-accurate fashion and accurately estimates such performance degradation.

Table 5: Total execution cycle predicted by Vivado HLS synthesis report and FLASH, and its error rate compared to the RTL-simulated result

Benchmark	RTL sim	Vivado HLS	FLASH
`Toy_mpath`	4,500,010	2,000,016	4,500,010
	-	(-56%)	(0%)
`Stencil`	524,309	524,299	524,309
	-	(~0%)	(0%)
`MD_sim`	12,089	10,498	12,089
	-	(-13%)	(0%)
`Mat_mul`	330,006	131,075	330,006
	-	(-60%)	(0%)
AVG	-	(-32%)	(0%)

8 CONCLUDING REMARKS

By simulating based on the scheduling information, we were able to solve the correctness issue of the software simulators and also provide accurate performance estimation. Also, simulating without allocation / binding information and component libraries allowed us to achieve three orders of magnitude faster speed compared to the RTL simulators. We have described an automated code generation flow that enables this new simulation flow.

We hope that the promising result presented in this work will motivate HLS commercial tool industry to provide additional routine that simulates based on the scheduling information only. This will substantially decrease the validation time of the customers who wish to rapidly estimate cycle-accurate performance, obtain correct output data, or detect possible deadlock situations.

As a future work, we will continue to widen the range of benchmarks so that the transformation flow will be robust enough to accommodate any Vivado HLS input code. We hope to include the Intel HLS flow if their tool's synthesis report provides detailed schedule information in the future. Also, we will enhance the output analysis stage to provide better functional and performance debugging support. In addition, we plan to add parallelization using Pthread/OpenMP so that large-scale simulation can be performed by exploiting multicore architecture.

ACKNOWLEDGMENTS

This research is partially supported by Intel and NSF Joint Research Center on Computer Assisted Programming for Heterogeneous Architectures (CAPA) (CCF-1723773). We are grateful to Xilinx for the software and the hardware donation. We thank Professor Miryung Kim (UCLA), Chaosheng Shi (Xilinx), and Professor Zhiru Zhang (Cornell Univ.) for the helpful discussions and the suggestions. We also thank Janice Wheeler for proofreading this paper.

REFERENCES

[1] AlphaData. 2017. Alpha Data ADM-PCIE-KU3 Datasheet. (2017). http://www.alpha-data.com/pdfs/adm-pcie-ku3.pdf
[2] A. Canis, et al. 2013. From software to accelerators with LegUp high-level synthesis,. In *Proc. Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems (CASES'13)*. 18–26.
[3] Y. Chi, J. Cong, P. Wei, and P. Zhou. 2018. SODA : stencil with optimized dataflow architecture. In *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD'18)*.
[4] Y. Choi and J. Cong. 2017. HLScope: High-level performance debugging for FPGA designs,. In *IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machines (FCCM'17)*. 125–128.
[5] Y. Choi, P. Zhang, P. Li, and J. Cong. 2017. HLScope+: Fast and accurate performance estimation for FPGA HLS. In *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD'17)*. 691–698.
[6] M. Chung, J. Kim, and S. Ryu. 2014. SimParallel: A high performance parallel SystemC simulator using hierarchical multi-threading. In *IEEE Int. Symp. Circuits and Systems (ISCAS'14)*. 1472–1475.
[7] J. Cong, Z. Fang, H. Kianinejad, and P. Wei. 2016. Revisiting FPGA acceleration of molecular dynamics simulation with dynamic data flow behavior in high-level synthesis. *ArXiv Preprint* (2016). http://https://arxiv.org/pdf/1611.04474.pdf
[8] J. Cong and J. Wang. 2018. PolySA: polyhedral-based systolic array auto compilation. In *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD'18)*.
[9] J. Cong, et al. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.
[10] P. Coussy, et al. 2009. An introduction to high-level synthesis. *IEEE Design & Test of Comput.* 26, 4 (2009), 8–17.
[11] S. Dai, M. Tan, K. Hao, and Z. Zhang. 2014. Flushing-enabled loop pipelining for high-level synthesis. In *Proc. Ann. Design Automation Conf. (DAC'14)*.
[12] J. Goeders and S. Wilton. 2015. Using dynamic signal-tracing to debug compiler-optimized HLS circuits on FPGAs,. In *IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machines (FCCM'15)*. 127–134.
[13] Intel. 2018. Intel Xeon Processor E5-2680 v4. (2018). www.intel.com/
[14] Intel. 2018. Intel FPGA SDK for OpenCL Pro Edition. (2018). https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf
[15] Intel. 2018. Quartus Prime Pro Edition Handbook. (2018). www.intel.com/
[16] J. Jang, S. Choi, and V. Prasanna. 2005. Energy-and time-efficient matrix multiplication on FPGAs. *IEEE Trans. Very Large Scale Integration* 13, 11 (2005), 1305–1319.
[17] S. Lahti, P. Sjövall, and J. Vanne. 2018. Are we there yet? A study on the state of high-level synthesis. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* (2018).
[18] J. Monson and B. Hutchings. 2014. New approaches for in-system debug of behaviorally-synthesized FPGA circuits,. In *IEEE Int. Conf. Field Programmable Logic and Appl. (FPL'14)*.
[19] ROSE. 2018. ROSE compiler infrastructure. (2018). http://rosecompiler.org/
[20] T. Schmidt, G. Liu, and R. Dömer. 2017. Exploiting thread and data level parallelism for ultimate parallel SystemC simulation. In *Proc. Ann. Design Automation Conf. (DAC'17)*.
[21] E. Sozzo, et al. 2017. A common backend for hardware acceleration on FPGA. In *IEEE Int. Conf. Comput. Design (ICCD'17)*. 427–430.
[22] A. Verma, et al. 2017. Developing dynamic profiling and debugging support in OpenCL for FPGAs. In *Proc. Ann. Design Automation Conf. (DAC'17)*. 56–61.
[23] Xilinx. 2018. UltraScale architecture and product data sheet: overview (DS890). (2018). https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf
[24] Xilinx. 2018. Vivado High-level Synthesis (UG902). (2018). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug902-vivado-high-level-synthesis.pdf
[25] C. Yu, et al. 2018. S2FA: an accelerator automation framework for heterogeneous computing in datacenters. In *Proc. Ann. Design Automation Conf. (DAC'18)*.