FLASH: Fast, ParalleL, and Accurate Simulator for HLS

Young-kyu Choi, *Member, IEEE*, Yuze Chi, Jie Wang, and Jason Cong, *Fellow, IEEE*

***Abstract*—A large semantic gap between a high-level synthesis (HLS) design and a low-level RTL simulation environment often creates a barrier for those who are not FPGA experts. Moreover, such a low-level simulation takes a long time to complete. Software HLS simulators can help bridge this gap and accelerate the simulation process; but their shortcoming is that they do not provide performance estimation. To make matters worse, we found that the current FPGA HLS commercial software simulators sometimes produce incorrect results. In order to solve these performance estimation and correctness problems while maintaining the high speed of software simulators, this paper proposes a new HLS simulation flow named FLASH. The main idea behind the proposed flow is to extract 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.**

***Index Terms*—Simulation acceleration, high-level synthesis, field-programmable gate array, source-to-source transformation.**

I. INTRODUCTION

ALTHOUGH the field-programmable gate array (FPGA) has many promising features that include 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 [1], [2] and Intel OpenCL HLS [3] 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 [4]–[6].

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 (e.g., ModelSim [7], NCSim [8], or VCS [9]) or on-board emulators (e.g., Zebu [10]) and try to map the result back to HLS. The result is often incomprehensible to those who are not FPGA experts. Moreover, low-level RTL simulation takes a very long time. Some work has been done to automate hardware probe insertion from the HLS source file [11]–[17], but this work requires regeneration of the FPGA bitstream if there is a change in the debugging point. The turnaround time is often in hours. On-board emulators also have a similar problem and require a long time for the bitstream generation.

These problems can be partially solved by the software-based simulators provided by HLS tools. The HLS software simulators compile the C or OpenCL source code for native execution on the host machine. 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.

An example can be found in the molecular dynamics simulation [18] (Fig. 1). Multiple distance processing elements (Dist PEs) filter out faraway molecules above threshold and send them to Force PE. The pruned molecules create a bubble (empty data) in the FIFO, and Force PE processes 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 software simulator finishes the simulation of Dist PE1 first, followed by Dist PE2, and so on. As a result, by the time the Force PE is simulated, the bubbles in the FIFOs are completely removed, and the Force PE output ordering can be entirely different from the RTL simulation. If one is trying to quickly trace the source of a problem that was observed in the output of an RTL simulation, the person will not be able to reproduce the problematic state in the software simulation.

Another problematic example can be found in the artificial deadlock situation [19], which occurs when the depth of the FIFO is smaller than the latency difference among modules (details in Section III-B). The issue is that the HLS software simulator cannot detect the deadlock situation and proceeds as if there is no problem with the design. We also have found a problem in the simulation of feedback loops where the feedback data is ignored by the HLS tool (Section III-C).

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 ([20]) commercial FPGA HLS tools, Xilinx Vivado HLS (VHLS) [2] and Intel OpenCL HLS (AOCL) [3], is presented in Table I. VHLS assumes unlimited FIFO depth, which makes it difficult to accurately model FIFO fullness/emptiness. Also, the sequential simulation execution

Manuscript received June 21, 2019; revised December 6, 2019; accepted January 7, 2020. Date of publication March 1, 2020; date of current version October 1, 2020. This paper was recommended by Associate Editor J. Cortadella. The authors are with Computer Science Department, University of California, Los Angeles, CA, 90095, USA. E-mail: {ykchoi,chiyuze,jiewang,cong}@cs.ucla.edu

This research is in part supported by Intel and NSF Joint Research Center on Computer Assisted Programming for Heterogeneous Architectures (CAPA) (CCF-1723773) and Baidu, NEC, and VMWare under the CDSC industrial partnership.

Digital Object Identifier XX.XXXX/TCAD.2020.XXXXXXX

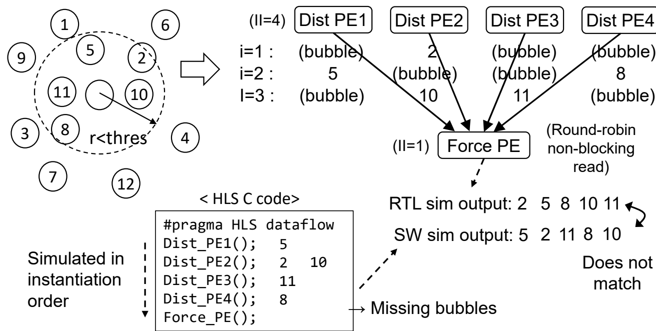


Fig. 1. Molecular dynamics simulation [18]

TABLE I
COMPARISON OF THE SOFTWARE SIMULATION OF XILINX VIVADO HLS [2] AND INTEL OPENCL HLS [3]. UNDESIRABLE CHARACTERISTICS ARE IN BOLD.

	Xilinx VHLS C Sim	Intel AOCL 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**

model prevents correctly simulating designs with feedback loops (Section III-C). AOCL simulates about 5X slower than VHLS, but it correctly simulates the FIFO depth. The tool assigns a thread to each module for concurrent simulation; but the execution order of the threads is not deterministic and may produce different results in different simulation runs for cases in Section III.

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, because it simulates low-level implementation details. We attempt to devise a new simulation flow that solves both problems. The idea is to add the scheduling information of C statements in the HLS software simulation. The new simulation flow would be faster than the RTL simulation without the allocation / binding information and the component libraries; it would solve the correctness problem of the software simulation and provide accurate performance estimation with its cycle accuracy.

Although simulating based on the LLVM intermediate representation (IR) is a possible option, we have instead decided to simulate in C syntax (augmented with scheduling information). This allows us to raise the simulation abstraction level—accelerating the simulation process and making it easier for programmers to understand what is being simulated. To our knowledge, this is the first HLS-based software simulation flow that takes such an approach.

By taking such an approach, however, several challenges were encountered (elaborated on in Section IV). One problem is how to guarantee cycle-accuracy of untimed C statements. Another is correctly simulating the parallelism that is inherent in hardware (and the corresponding RTL simulation) in sequential C semantics.

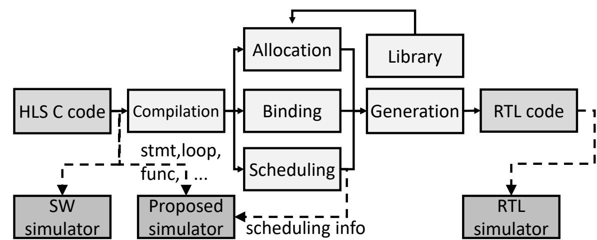


Fig. 2. HLS design steps [21] and comparison of simulation flows

In this paper, we propose FLASH—an HLS software simulation (HSS) flow that addresses these challenges. We describe transformations that allow cycle-accurate simulation of FIFO communication (defined in Section IV). Also, a method is presented to simulate task-level and pipelined parallelism with C semantics. These steps are described in Section V.

In order to simulate pipelined parallelism, variables need to be duplicated to match the depth of a loop pipeline (explained in Section V-B1). But this results in a redundant data copy, which slows down the simulation. We propose optimization techniques to reduce this overhead in Section VI.

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 is made to be compatible with the conventional HLS software simulator for easy integration with the existing tool. The overall flow is described in Section VII.

FLASH also provides correctness and performance debugging support for programmers. In order to ease the process of detecting deadlocks or stalls, a set of source-level directives is included. Also, the debugging time is shortened by allowing variables to be added to the capture list in the middle of simulation. This will be explained in Section VIII.

The contribution of this paper can be summarized as follows:

- We show that simulating based on the scheduling information can help solve the correctness issue of HLS software simulators and rapidly provide accurate performance estimation.
- We develop a framework that allows fast cycle-accurate simulation of an HLS design. Several code transformation techniques have been presented to enable this process. Moreover, optimizations are proposed to accelerate the simulation speed.
- We propose unique debugging features for HSS.

This paper is an extended version of our preliminary work presented in [22]. Compared to [22], this paper presents new optimizations to accelerate the simulation speed (Section VI). We also propose novel source-level debugging features (Section VIII). Moreover, we add a formal definition of the problem and a detailed explanation of the proposed solution and the code transformation process (Sections III, IV, and V).

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

II. Related Work

Work in [11]–[17] 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 is more focused on verifying functional correctness [11]–[15] and work that is more focused on extracting performance-related parameters [16], [17]. Goeders and Wilton describe how to record and replay the value of variables from an HLS-generated circuit [11]. Their work is extended to cover compiler-optimized designs in [12] and to allow offline signal restoration in [13]. Monson and Hutchings introduce event observability ports (EOP) to enable source-level signal trace and explain how to combine multiple signals to reduce trace buffer size [14], [15]. HLScope [16] describes an in-FPGA monitoring flow that extracts cycle information from FPGA designs written in C. The work of Verma et al. [17] is based on OpenCL and measures stall latency and monitors memory access patterns by utilizing trace buffers to store an event's timestamp. However, these hardware-based HLS debuggers typically require hours of initial overhead for bitstream generation.

There are other software-based HLS simulators. The LegUp HLS [23] simulator provides a speedup prediction based on the profiling result of the source code and the execution cycle from its synthesis result. HLScope+ [24] describes a method to extract cycle information that is hidden by HLS abstraction and uses VHLS C simulation to predict the performance for applications with dynamic behavior. These works, however, do not guarantee cycle-accuracy.

There are several SystemC simulators (e.g., [25]–[27]) that achieve cycle-accuracy for the source code that has explicit scheduling information specified by the programmer. However, constructing a cycle-accurate input design file may be too difficult for non-experts. Our flow, on the other hand, achieves cycle-accuracy for an HLS C source code without requiring user-defined scheduling information.

There is a class of work that accelerates the simulation of an HLS tool's output RTL code by converting the RTL code into a cycle-accurate C model [28], [29]. Mahapatra et al. [28] report a speedup of 5X after removing the core computation and only maintaining the IO timing, but such an approach cannot be used for data-dependent benchmarks. Verilator [29], on the other hand, can be used to provide a functionally correct and cycle-accurate HLS simulation as our work. Verilator employs several techniques for acceleration—such as removal of time delays, randomized unknown value, and creation of table lookups. But the speedup in Verilator is limited because it is very difficult to *completely* remove allocation and binding information from the RTL code—whereas in our approach, this information is never added in the first place. A quantitative comparison is presented in Section IX.

III. Problem Description and Motivating Examples

In this section, we describe four classes of problems (three correctness-related and one performance-related) in current HLS software simulators. The problems are demonstrated with relevant examples in the literature.

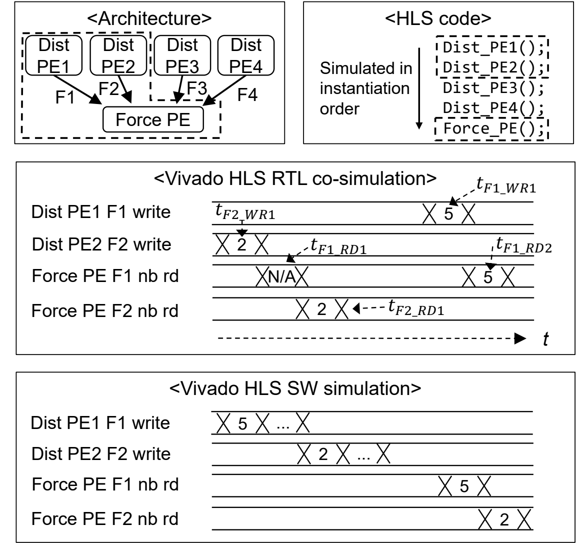


Fig. 3. Timing diagram of the molecular dynamics simulation in Fig. 1 (FIFO transactions among only Dist PE1, Dist PE2, and Force PE are shown)

A. Data Ordering Problem

The problem of incorrect output ordering in the HSS for molecular dynamics simulation was presented in our introduction. In this section we discuss the cause of this problem in more detail. Fig. 3 shows the timing diagram of the FIFO transactions among Dist PE1, Dist PE2, and Force PE in Fig. 1. Dist PE1 and Dist PE2 communicate with Force PE through FIFO F1 and F2 respectively. Consider a case where data (2) is written to F2 before data read from F1 and F2, and F1 is written (data 5) afterwards (illustrated in the RTL simulation part of Fig. 3). At the time of the first F1 non-blocking read attempt (t_{F1_RD1}), the first F1 write (t_{F1_WR1}) has not yet occurred ($t_{F1_RD1} < t_{F1_WR1}$), and the successful F2 read precedes the successful F1 read ($t_{F2_RD1} < t_{F1_RD2}$).

In the VHLS software simulation, however, data (5) is available in the first read attempt to F1 because Dist PE1 is evaluated entirely before Dist PE2 and Force PE. That is, unlike the RTL simulation, the first F1 write has already occurred before the first F1 read attempt ($t_{F1_RD1} > t_{F1_WR1}$). As a result, the successful F2 read happens after the successful F1 read ($t_{F2_RD1} > t_{F1_RD2}$). The ordering of the data processed at Force PE is not maintained. If the HSS has evaluated the FIFO write correctly before each FIFO read attempts (i.e., $t_{F1_RD1} < t_{F1_WR1} < t_{F1_RD2}$ and $t_{F2_WR1} < t_{F2_RD1}$), this problem would not have occurred. In the AOCL simulation, the simulation order of the producer modules is undetermined, and a similar data ordering problem occurs.

As demonstrated in the example, the HLS software simulator should evaluate the FIFO writes before each non-blocking read attempt in the same order as in the RTL simulation. If not, a data ordering problem may occur. The *data ordering problem* is defined as a case where a consumer module M_C is reading data in a non-blocking fashion from multiple producer modules M_P through FIFOs, and the order of data processed at M_C in the RTL simulation is not maintained in the HSS.

B. Module Latency Problem

Consider an example in Fig. 4 (named `toy_mpath`) 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`. Because of back pressure from `M2` and `fifo3`, `fifo1` becomes full. Then `M1` stops producing output to `fifo2` because `fifo1` and `fifo2` have to be written in the same cycle. `fifo2` eventually becomes empty, which blocks the pipeline of `M3`. Even though `M3` has consumed some remaining data in `fifo2`, `fifo4` is still empty because of `M3`'s long latency. Then none of the modules can do any further useful work, and the circuit deadlocks. This is called an artificial deadlock. The artificial deadlock is caused by the mismatching latency of multiple datapaths and inadequate FIFO depth to balance the latency difference [19]. We can also observe this in the architecture for stencil computations [30] that contains modules and FIFOs with various latencies and depths.

In order to reproduce the deadlock situation, an HLS software simulator should create the output data after reading input with a delay that reflects the module latency. However, existing HLS software simulators evaluate each iteration of a loop as if the data is instantaneously passed from input to output. Thus, the latency among different datapaths is not simulated, and the artificial deadlock does not occur. As a result, even after running a HSS, the user is unaware of a potential problem that might occur during actual on-board execution.

We will refer to this problem as the *module latency problem*. Suppose that a module has a sequence of C FIFO read and write statements $cstmt_1$, ... $cstmt_c$, ... $cstmt_C$. If $cstmt_c$ is a blocking read/write, multiple read/write attempts may be performed before the read/write is completed. If non-blocking, read/write is always completed on the first attempt. We assume that the HLS tool has scheduled a delay of $delay_c$ cycles between the completion of $cstmt_c$ and the first attempt of $cstmt_{c+1}$. This delay reflects the computation latency. The module latency problem is defined as a case where HSS fails to simulate $delay_c$ between the first attempt of $cstmt_{c+1}$ and the completion of $cstmt_c$ for some of $c = 1 \ldots C-1$.

Note that we have modified `M2` to the code shown in Fig. 5. The purpose is to make a fair comparison of the simulation time by making the module simulation to finish at the same point (the HLS RTL simulation of Fig. 4 will deadlock, whereas HSS will not). If the input FIFO is empty, a bubble is inserted into the pipeline (line 4 of Fig. 5)—this allows the pipeline to keep processing the already-read data even if there is no additional input. A similar transformation is applied on `M3`. Deadlock situation does not occur, [1] because `M4` can now receive the output from `M3`.

[1] Alternative solutions for deadlock avoidance are presented in [19], but they require modification to the HLS scheduling, allocation, and binding kernels. Deadlock can also be avoided by increasing the buffer size [31] (AOCL has this functionality). However, since the efficiency of the solution for avoiding deadlock is not the focus of this work, we apply a solution that only requires simple source-to-source transformation of the loops without an elaborate analysis of the whole circuit. This method cannot be used to resolve all deadlocks—such as the one that is caused by circular wait.

C. Feedback Problem

As mentioned previously, the VHLS software simulator evaluates functions in the order they are instantiated in the source code. This causes a problem if there is a feedback path that passes data from later instantiated functions to earlier ones. At the time earlier functions are simulated, the data is not be available. As a result, VHLS simulates the program as if the FIFOs in the feedback path are always empty. We will refer to this issue as a *feedback problem*. The feedback problem occurs when the content of a FIFO buffer in the HSS does not match that in the RTL simulation at the cycle a read operation is performed on a FIFO in a feedback path. This happens when FIFO writes before each read is not correctly simulated. An example of the feedback problem in the case of matrix multiplication can be found in [22]. The AOCL tool can simulate the feedback data from a blocking read correctly because a thread simulating each module can wait for others to pass the data. However, it is not guaranteed that the feedback data from a non-blocking read will arrive at the right timing.

D. Performance Estimation Problem

Performance estimation problem is defined as providing incorrect estimation of the module execution time. VHLS synthesis report has a performance estimation problem for applications with data-dependent loop bounds, conditional statements, or stalls [32], and almost all benchmarks used for the experiment have such properties (details in Section IX-C). AOCL synthesis report and VHLS/AOCL software simulators do not provide any performance estimation at all.

IV. Problem Statement and Challenges

Before we provide the problem statement, we will define the concept of *FIFO communication cycle-accurate* (FCCA) simulation. The FIFO communication refers to the FIFO-accessing expressions in the source code (listed in the second column of Table II). A FIFO communication statement refers to a statement with a FIFO communication. Let us assume that a FIFO communication has been evaluated in HSS at cycle t. We declare that the FIFO communication is simulated *cycle-accurately* if the FIFO input value and the FIFO output value of the FIFO APIs (FAPIs) in the HSS match the FIFO input ports (`din`, `rd_en`, `wr_en`) and the FIFO output ports (`dout`, `empty`, `full`) [33] in RTL simulation at the same cycle t. That is, the C variables and the RTL signals have the same value as described in the third column of Table II at the same cycle t. The FIFO input value of the FAPIs refers to the value of "wdata", and the FIFO output value of the FAPIs refers to the value of "test" and "rdata" in Table II. If all FIFO communication in a C source code is simulated cycle-accurately in HSS, the simulation will be FCCA.

For example, the non-blocking read expression, "test = fifo.`read_nb`(rdata)", is cycle-accurately simulated if the value of "rdata" matches `dout` and "test" has the toggled value of `empty` at the same cycle as the RTL simulation.

We assume that we simulate an HLS design that is composed of multiple finite-state machine (FSM) modules (inferred from C functions). The modules execute concurrently

TABLE II
THE FIFO COMMUNICATION IN THE C SOURCE CODE, THE CORRESPONDING FIFO IP RTL PORTS AND C VARIABLES, AND THE CORRESPONDING FLASH SIMULATION CODE (VHLS FIFO APIS [2] AND FIFO IP RTL PORTS [33] ARE IN `MONOSPACE FONT`)

Description	FIFO comm in source code	RTL ports & C variables	FLASH simulation code
Blocking read	rdata = fifo.`read`()	1 == !`empty`, `rd_en` = 1, rdata = `dout`	(stall cond: fifo_rnum==0), test = (..rnum!=0), rdata = fifo_arr[fifo_rptr++]; fifo_rnum−−;
Non-blocking read	test = fifo.`read_nb`(rdata)	test = !`rd_en` = !`empty`, rdata = `dout`	
Blocking write	fifo.`write`(wdata)	1 == !`full`, `wr_en` = 1, `din` = wdata	(stall cond: fifo_wnum==0), test = (..wnum!=0), fifo_arr[fifo_wptr++] = wdata; fifo_wnum−−;
Non-blocking write	test = fifo.`write_nb`(wdata)	test = !`wr_en` = !`full`, `din` = wdata	
Empty	test = fifo.`empty`()	test = `empty`	test = (fifo_rnum == 0)
Full	test = fifo.`full`()	test = `full`	test = (fifo_wnum == 0)

(with directive `#pragma HLS dataflow`) and use streaming FIFOs for inter-module communication.

Our main goal is to construct an HLS software simulator that is FCCA. The input and output of the simulator is defined as follows:

Input: (1) An HLS C source code (2) Scheduling information (3) Input data of the design

Output: Output data of the design

The scheduling information is defined as the information on the FSM state transition and the assigned FSM state of FIFO communication, conditional statements, and loop statements.

FCCA simulator does not have the data ordering, module latency, feedback, and performance estimation problems that were described in Section III.

Recall that the data ordering problem occurs when a consumer module M_C is reading data in a non-blocking fashion from multiple producer modules M_P through FIFOs and the order of data processed at M_C is not maintained in the HSS. If the FIFO communication is cycle-accurate, the relative ordering of FIFO reads and writes matches that of the RTL simulation. That is, the number of FIFO reads and writes and the data before each FIFO read match that of the RTL simulation. Thus, the order of data read from the FIFO at M_C in FCCA simulation matches that of the RTL simulation. The proof that an FCCA simulator does not have the data ordering problem is provided in [32].

We have explained that the feedback problem happens with incorrect simulation of writes before each read operation to a FIFO in the feedback path. Since the relative ordering of reads and writes is maintained in an FCCA simulator, the feedback problem does not take place.

Since all FIFO transactions occur at the same cycle as in the RTL simulation, the delay between any consecutive pair of FIFO reads and writes of a module matches that of the RTL simulation. Thus, the FCCA simulator does not have the module latency problem.

Let us assume that a simulator can model modules' FSM states correctly if stalls caused by `empty` and `full` FIFO signals have been simulated correctly. We also assume that a module's execution time is determined by its FSM state (more explanation on these assumptions in Section V-A2). Since FCCA simulator models the `empty` and `full` signals cycle-accurately, it estimates the modules' execution time accurately and does not have the performance estimation problem.

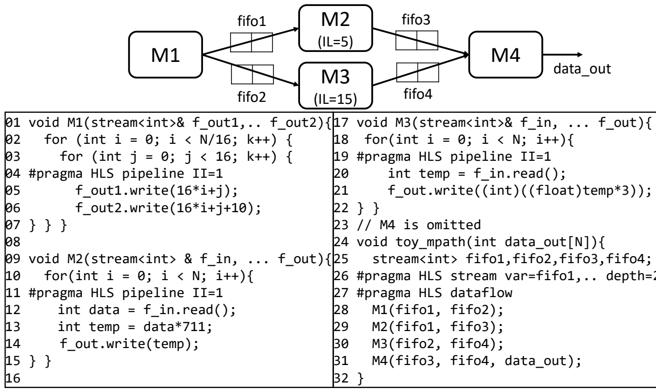
In addition to the main goal of achieving cycle-accurate FIFO communication, the simulator should provide the content of the registers (e.g., the state of a module or the number of empty FIFO buffers) in a deadlock situation for debugging purposes. Moreover, the simulation code should be semantically 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: FCCA simulation**
  It is difficult to discover the exact cycle when statements are executed since the information given by the HLS tool is very limited. For example, The AOCL tool only provides loop initiation intervals (II). The Vivado HLS tool provides slightly more information—it provides a list of LLVM IR and the corresponding state of an FSM. But mapping such low-level representation (e.g., lines 27–31 of Fig. 6) back to the original C code is a difficult task. Moreover, the execution cycle may change due to FIFO being empty or full. Even if the execution cycle is known, an FCCA simulator needs to *selectively* simulate a code region that corresponds to a particular cycle. Furthermore, the value of variables at a certain cycle must be correctly supplied to the simulation of the next cycle.
- **Challenge 2: Simulation of parallelism**
  HLS designs have multiple levels of parallelism including task-level parallelism and pipelined parallelism. Cycle-accurately simulating parallelism in a C syntax becomes a difficult task because the value of variables and the simulation order of the statements become different from that of the source code. For example, if the statement in line 21 of Fig. 4 is executed 14 cycles after the statement in line 20, we would need to simulate line 21 with a "temp" value that corresponds to iteration i and line 20 with that of iteration $i + 14$ in a single cycle.
- **Challenge 3: Loop and function simulation**
  We need to construct an equivalent model of high-level C semantic such as loops and functions.

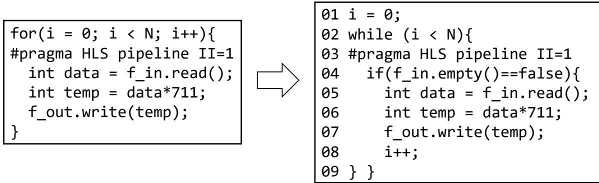
V. AUTOMATED CODE GENERATION FOR RAPID CYCLE-ACCURATE SIMULATION

In this section we provide a solution to each challenge in Section IV and describe our proposed automated simulation code generation flow. For illustration, we use the `toy_mpath` example (Fig. 4) after modifying the source code to avoid the deadlock as shown in Fig. 5.




```
01 void M1(stream<int>& f_out1,.. f_out2){
02   for (int i = 0; i < N/16; k++) {
03     for (int j = 0; j < 16; k++) {
04 #pragma HLS pipeline II=1
05       f_out1.write(16*i+j);
06       f_out2.write(16*i+j+10);
07 } } }
08
09 void M2(stream<int> & f_in, ... f_out){
10   for(int i = 0; i < N; i++){
11 #pragma HLS pipeline II=1
12     int data = f_in.read();
13     int temp = data*711;
14     f_out.write(temp);
15 } }
16
17 void M3(stream<int>& f_in, ... f_out){
18  for(int i = 0; i < N; i++){
19 #pragma HLS pipeline II=1
20     int temp = f_in.read();
21     f_out.write((int)((float)temp*3));
22 } }
23 // M4 is omitted
24 void toy_mpath(int data_out[N]){
25   stream<int> fifo1,fifo2,fifo3,fifo4;
26 #pragma HLS stream var=fifo1,.. depth=2
27 #pragma HLS dataflow
28   M1(fifo1, fifo2);
29   M2(fifo1, fifo3);
30   M3(fifo2, fifo4);
31   M4(fifo3, fifo4, data_out);
32 }
```

Fig. 4. Structure and code for motivating example `toy_mpath`



```
for(i = 0; i < N; i++){
#pragma HLS pipeline II=1
  int data = f_in.read();
  int temp = data*711;
  f_out.write(temp);
}
```

```
01 i = 0;
02 while (i < N){
03 #pragma HLS pipeline II=1
04   if(f_in.empty()==false){
05     int data = f_in.read();
06     int temp = data*711;
07     f_out.write(temp);
08     i++;
09 } }
```

Fig. 5. Modified code of `M2` in Fig. 4 to avoid artificial deadlock

```
01 ==========================================================
02 + Verbose Summary: Schedule
03 ==========================================================
04 * Number of FSM states : 7
05 * Pipeline : 1
06   Pipeline-0 : II = 1, D = 5, States = { 2 3 4 5 6 }
07 * Dataflow Pipeline: 0
08
09 * FSM state transitions:
10 1 -->
11         2  / true
12 2 -->
13         7  / (!tmp)
14         3  / (tmp)
15 3 -->
16         4  / true
17 4 -->
18         5  / true
19 5 -->
20         6  / true
21 6 -->
22         2  / true
23 7 -->
24
25 * FSM state operations:
26
27 State 6 <SV = 5> <Delay = 1.75>
28 ... Operation 27 ... ---> "call void (...)* @_ssdm_op_Spec
29 ... Operation 28 ... --->   "call void @_ssdm_op_Write
.ap_fifo.volatile.i32P(i32* %f_out_V, i32 %temp)"
30 ... Operation 29 ... --->   "br label %._crit_edge ...
31 ... Operation 30 ... --->   "%empty_40 = call i32 ...
```

Fig. 6. VHLS scheduling report for `M2` of Fig. 5

```
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     M2_state = 2;                 //state transition for state 1
07   }
08   else if(M2_state == 2){ //state conditional block for state 2
09     ...           //computation stmt & communication for state 2
10     M2_state = 7;                 //state transition for state 2
11   }
12 }                 //exit sim function after simulating one cycle
```

Fig. 7. Simulation function structure for selective simulation of an FSM state (M2_SIM is simulated at line 9 of Fig. 9)

```
01 static bool p1_en_st3, ... p1_en_st6  = false; //enable signals
02 static int temp_st3, ... temp_st6; //6    //pipelined variables
03 ...
04 else if(M2_state == 2){ //starting state for the pipelined loop
05   if(p1_en_st6 && fifo3_wnum==0){//if stalled due to FIFO3 full
06     return;         //exit without any changes (see Sect 5.A.4)
07   }
08   ...
09   if( p1_en_st6 == true ){   //enabled 4 cycles after FIFO read
10     p1_en_st6 = false;       //disables enable signal after use
11     fifo3_arr[fifo3_wptr++] = temp_st6; //7   //FIFO data write
12     fifo3_wnum--;                             //(see Sect 5.A.3)
13   }
14   ...
15   if( p1_en_st3 == true ){    //enabled 1 cycle after FIFO read
16     p1_en_st3 = false;       //disables enable signal after use
17     p1_en_st4 = true;              //enable signal propagation
18     temp_st4 = temp_st3;  //copies variable for next pipe stage
19   }
20   if( i_st2 < N ){ //2     //loop exit condition (see Sect 5.C)
21     if( fifo1_rnum != 0 ){ //4             //if FIFO not empty
22       data_st2 = fifo1_arr[fifo1_rptr++]; //5  //FIFO data read
23       fifo1_rnum--;                           //(see Sect 5.A.3)
24       temp_st2 = data_st2 * 711; //6 // comp stmt mapped to st2
25       i_st2++; //8       // loop iterator update (see Sect 5.C)
26       p1_en_st3 = true; //enables if path for later pipe stages
27       temp_st3 = temp_st2;//copies variable for next pipe stage
28 ... } } }
```

Fig. 8. Simulation code that models pipelined loop parallelism for `M2` of Fig. 5 (provides details for line 9 of Fig. 7)

```
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] = F1_SIM;        .... Flist[3] = F4_SIM;
05
06 while(1){                                  //scheduler loop
07   ...   // loop until until deadlock or all modules finish
08   for(x = 0; x < M; x++)          //simulate all modules
09     Mlist[x]();
10   for(p = 0; p < F; p++)            //simulate all FIFOs
11     Flist[p]();
12   ...
13   cycle++;
14 }
```

Fig. 9. Module/FIFO simulation scheduler to model task-level parallelism

A. FIFO Communication Cycle-Accurate Simulation

We will describe the properties of FLASH and the corresponding code transformation. Based on these properties, we will explain how FLASH achieves FIFO communication cycle-accurate (FCCA) simulation.

1) Matching Simulated State of Statements: Let us assume that HLS tool schedules a FIFO communication of a C source code to be executed at a particular FSM state (st) of a module. FLASH simulates the FIFO communication at the same st scheduled by the HLS tool. In order to achieve this, we first need to obtain the HLS scheduling information of the FIFO communication. This is found from parsing FIFO-related keywords in the scheduling report. For example, the state when FIFO “f_out” performs the write operation (line 7 of Fig. 5) is found to be 6, because `op_Write.ap_fifo` and “f_out” keywords are detected in line 29 of Fig. 6. Similarly, the FIFO read statement (line 5 of Fig. 5) is assigned to state 2 from the scheduling report (not shown in the figure).

Next, we need to ensure that only the FIFO communication statements assigned to each FSM state are selectively simulated at every cycle. We declare an FSM state variable (line 2 of Fig. 7) for each module and copy statements to

the conditional block that correspond to its simulated state (*state conditional block*). An example can be found for the `M2` module in lines 4–7 ($st = 1$) and lines 8–11 ($st = 2$) of Fig. 7. After the simulation function of a module has been called, only the statements for a single FSM state are simulated, and then the function exits. That is, a single clock event is simulated by a function entrance and exit.

Since FLASH aims for cycle-accuracy of FIFO communication, the computation statements do not need to be evaluated cycle-accurately. For computation statements, we can assign an arbitrary state as long as it does not violate the timing causality with the cycle-known FIFO communication that has dependency with the computation statement. We group the computation statements to a few FSM states as much as possible; if the statements are spread among multiple FSM states, the variables shared across the states may need to be stored in cache or DRAM and loaded back after function exit and entrance. This is inefficient if the variable has a short life and could have been optimized to a CPU register.

For example, the computation statement in line 6 of Fig. 5 has a dependency with both the FIFO read and the FIFO write statements. It may be assigned to any state between 2 and 6 without violating the time causality, but to reduce the number of FSM states with statements, it should be assigned to either 2 or 6. We choose to assign it to state 2, following the as-soon-as-possible scheduling policy, as it tends to reduce the number of variables being passed between the states.

2) Cycle-Accurate FSM State: FLASH cycle-accurately simulates the FSM state of a module at t (st_t). By induction, st_t is cycle-accurate if the initial state at $t = 1$ is known ($st_{t=1} = 1$) and the state transition Δ_t matches the RTL simulation at 1, 2, ..., t-1. Δ_t matches the RTL simulation if the state transition information can be obtained from the HLS tool report and a state transition statement that reflects this information is evaluated at t. Also, Δ_t should be stalled if `empty` or `full` signals have been asserted when the blocking reads or writes have been evaluated.

VHLS provides the state transition information in its scheduling report. For example, the loop in module `M2` in Fig. 4 is evaluated in states 2 to 6, as shown in line 6 of Fig. 6. The state transition of the loop is composed of intra-loop state transition (e.g., state 2 to 3, as shown in line 14 of Fig. 6) and loop exit (e.g., state 2 to 7, as shown in line 13). FLASH obtains this information and inserts the state transition statement into the simulation code. For example, line 10 of Fig. 7 reflects the loop exit state transition from state 2 to 7. The method used by FLASH to correctly simulate the state transition stalls will be discussed in Section V-A4.

VHLS schedules a module to start and finish its execution at a particular FSM state. Since FLASH cycle-accurately simulates the FSM state of a module, the estimation of a module's execution time is cycle-accurate.

3) FIFO Behavior Modeling: In the FLASH simulation code, the FIFO is implemented as a circular buffer with read/write pointers (fifo_rptr and fifo_wptr) and an array (fifo_arr). The array length is set to FIFO buffer size (FIFO_SIZE) plus one because one buffer space is kept empty in circular buffer implementation [34]. Also, we declare fifo_rnum and fifo_wnum variables to denote the number of data and buffer spaces available in the FIFO. FAPIs in the source code are transformed based on the fourth column of Table II. An example is shown in Fig. 8, which is the transformed simulation code from `M2` in Fig. 5. Line 7 of Fig. 5 is transformed to: "fifo3_arr[fifo3_wptr++] = temp_st6; fifo3_wnum−−;" (lines 11-12 of Fig. 8). The code difference between the blocking and the non-blocking FIFO communication is that the code for blocking communication is not simulated if the stall condition is satisfied. This is further explained in Section V-A4.

In addition to decreasing the number of buffer spaces (fifo3_wnum−−) for FIFO write, we would need to increase the number of available data (fifo3_rnum++). But this process is delayed until all other statements in the current cycle have been simulated. The reason is to match the Xilinx FIFO IP behavior [33] of allowing data in FIFO to be available for read, one cycle after it has been written. The details of this delayed processing is further provided in Section V-B2.

4) FSM Stall Modeling: FLASH cycle accurately models the FSM stalls due to FIFO being full or empty. If a stall condition is met, none of the statements of current FSM state should be simulated, and the simulation function should exit. To achieve this, the stall condition is placed at the beginning of a state conditional block. The simulation code for the stall condition is "fifo_rnum==0" for FIFO empty and "fifo_wnum==0" for FIFO full (Table II). These codes are also used for the stall conditions of the blocking reads and writes. For example, the stall condition that corresponds to the FIFO blocking write in line 7 of Fig. 5 is "if(p1_en_st6 && fifo3_wnum==0)". This condition has been added to line 5 of Fig. 8. Also, the function return statement has been added to line 6 of Fig. 8. Note that we add an enable signal "p1_en_st6" to the stall condition of a pipelined loop because the FIFO write occurs at FSM state 6 (more details in Section V-B1).

FLASH can detect a deadlock by checking if a state transition did not occur (stalled) in all modules. It is enabled by the source-level trigger directive that is explained in Section VIII-B.

It is worth noting that applying the classic event-driven simulation approach (e.g., [35]) makes little difference in the simulation speed of FLASH. The reason is that the stall condition is placed at the beginning of a state conditional block and prevents most of the statements from being evaluated when a module is stalled. That is, there is little overhead in processing a module without an event that requires simulation, and this diminishes the benefit of applying the event-driven approach.

5) Correctness of the Variable Reference: As explained in Section V-A1, all statements that have dependency with $cstmt$ have been evaluated before the simulation of $cstmt$. The value of variables written by statements with the same st as $cstmt$ is correctly supplied to $cstmt$ because they are simulated in the same state conditional block. A problem occurs when reading variables written by statements with FSM states other than st because the simulation function exits after each cycle. This problem is solved using the `static` keyword in variable declaration (e.g., line 2 of Fig. 7 and line 2 of Fig. 8). By

using this technique, the contents of the variables are restored and saved regardless of the simulation function entrance or exit.

6) Proof of FCCA Simulation: We can prove that FLASH is an FCCA simulator by demonstrating that the FIFO input and output values of the FAPIs in the FLASH simulation match the values of FIFO input and output ports in the RTL simulation at all clock cycles. Due to the space limitation, only a brief explanation of the proof will be provided—the details can be found in [32].

In FLASH, $cstmt$ is simulated at the same cycle t as the RTL simulation because the simulated st of $cstmt$ matches the HLS tool (Section V-A1) and st is simulated cycle-accurately (Section V-A2 and Section V-A4). $cstmt$ produces the correct value for the FIFO input value of the FAPIs if FLASH supplies the variables referenced by $cstmt$ (achieved by Section V-A5) and the FIFO output value of the FAPIs that match the RTL simulation. Correct FIFO output values of the FAPIs can be supplied at cycle t with accurate FIFO behavior modeling (Section V-A3) and the FIFO input value of FAPIs that matches the RTL simulation at cycle 1, 2, ... $t-1$. Then we can prove that FLASH is an FCCA simulator by induction.

B. Simulation of Parallelism

1) Pipelined Parallelism: At each cycle, all statements in a pipelined loop should be simulated in a pipelined parallelism fashion. The number of FSM states to be simulated corresponds to the loop iteration latency (IL, also called pipeline depth). If we simulate only a particular FSM state conditional block of a pipelined loop, it would not be possible simulate this parallelism.

To solve this problem, we would need to simulate all FSM states of a pipelined loop. It is possible to make an exception to the simulation structure by traversing through multiple state conditional blocks in a single cycle for pipelined loops; but this would over-complicate the simulation structure. For a simpler solution, we choose to move all of the pipelined loop's state conditional blocks into the conditional block of a single state. The reallocated conditional blocks are referred to as *pipeline stage conditional blocks*. As shown in Fig. 8, the contents of FSM states 2, 3, and 6 have been moved to pipeline stage conditional blocks in lines 20-28, lines 15-19, and lines 9-13.

If a pipelined loop L's II (II_L) is larger than 1, FLASH makes II_L state conditional blocks for this loop. In this case, the pipeline stage conditional blocks for state st are placed at state $(st_L + ((st - st_L)\%II_L))$ conditional block, where st_L is L's first FSM state.

We introduce *enable signal* to decide if the statements inside each pipeline stage conditional block will be evaluated. For example, the FIFO write at lines 11-12 of Fig. 8 is evaluated if the enable signal "p1_en_st6" at line 9 is one. Using an enable signal allow us to selectively simulate statements in a pipelined loop's prologue/epilogue and invalidate statements in a pipeline bubble (from the artificial deadlock avoidance transformation in Section III-B). The enable signal is also used to selectively simulate statements of a conditional block. The value of enable signals is propagated through the pipeline stages as shown in line 17.

It is important to note that the order of each pipeline stage conditional block has been *reversed* (st6, ... st3, st2). This limits the value of enable signals to be copied only to the immediate next pipeline stage in simulation of a single cycle.

Even if a same variable is used in different statements of the original source code, we cannot assume that they have the same value if they have been assigned to different pipeline stage conditional blocks. For example, suppose that line 6 of Fig. 5 is performed at FSM state 2, and line 7 is performed at state 6. In a single cycle of the pipelined loop simulation, "temp" of line 7 corresponds to loop iteration *i*, whereas "temp" of line 6 corresponds to loop iteration *i+4*. Thus, they would have different values.

For correct simulation, we keep multiple copies of the same variable for each pipelined stage of a loop. The variables are copied through the pipeline like shift registers. For example, the "temp" variable is copied from loop pipeline stage 3 to stage 4 at line 18 of Fig. 8. Variables "data" and "i" are not copied to the next pipelined stage after performing cycle-based variable liveness analysis (explained in Section VI-A). Similar to the enable signals, the content of pipelined variables is only copied to the immediate next state in a single cycle since the order of the pipeline stage conditional block has been reversed. Optimization of the pipelined variables is discussed in Section VI.

Because of the duplicated pipelined variables, the readability of the simulation code can be reduced. In order to diminish this side effect, FLASH places the line number of the original variable declaration in the source code as a comment of the duplicated pipelined variable declaration in the simulation code. For example, the line number 6 of the variable declaration of "temp" in Fig. 5 is written as a comment of the duplicated pipelined variable declaration in line 2 of Fig. 8. The original line numbers of the computation and communication statements are also placed at the comments of the simulation statements (e.g., lines 20-25 of Fig. 8).

2) Task-Level Parallelism: As discussed in Section V-A1, the statements in an FSM state are simulated by calling the simulation function of a module. Thus, the task-level parallelism can be simulated by calling all simulation functions in a round-robin fashion. This is processed in the *module simulation loop* shown in lines 8-9 of Fig. 9.

As mentioned in Section V-A3, the update of the buffer spaces and the number of available data is delayed until all modules in the current cycle have been simulated. The update (corresponding code is presented in [32]) is performed in the *FIFO simulation loop* (lines 10-11 of Fig. 9).

The module simulation loop and the FIFO simulation loop form the *scheduler loop* as shown in lines 6-14 of Fig. 9.

C. Loop and Function Simulation

The loop initialization statement of loop L is simulated upon initial entrance to L's first FSM state (st_L). If L is a pipelined loop, the enable signal is set to 0 at st_L (Section V-B1) when L's loop iterator of a new iteration does not satisfy the loop condition. Since the loop condition for an iteration should be checked after the loop update statement has been evaluated,

the loop update is evaluated just before transitioning into st_L. Recall that L has a number of state conditional blocks that matches II (II_L) of the loop (Section V-B1)—thus, the loop update (e.g., line 8 of Fig. 5) is evaluated at the end of ($st_L + II_L - 1$).

Since the loop update is evaluated before the final state of a loop, a dependency problem may occur. For example, suppose that we add a statement between line 7 and line 8 of Fig. 5 that is dependent on line 7 and references i. The loop index update statement in line 8 is scheduled to state 2 ($\because st_L = 2, II_L = 1$). Assuming line 7 is scheduled to state 6 from the scheduling report, the new statement between lines 7 and 8 incorrectly references i that has already been updated to the next iteration. This is solved by copying i to a temporary variable before evaluating the loop index update statement and renaming any reference of i that has the dependency problem to this temporary variable.

The state transition for pipelined loop exit occurs when the loop condition is not satisfied and all enable signals in the pipeline have been invalidated. Simulation of statements inside a pipelined loop has been discussed in Section V-B1. The code transformation method of a flattened loop can be found in [22].

A function call is simulated by sending a module enable signal to the scheduler loop (Fig. 9). Next, the function argument values are copied into the newly called module.

VI. Optimization of Pipelined Loops Simulation

Pipelined loops typically account for most of the execution time of many FPGA designs. To simulate pipelined parallelism, we need copies of variables that correspond to the loop's IL (Section V-B1). However, a naive implementation could lead to making redundant copies of the variables. This section discusses how to optimize this routine. The effect of the optimization will be presented in Section IX-B.

A. Cycle-Based Variable Liveness Analysis

The pipelined variables are only needed in the pipeline stages where the variables are being accessed. To ensure this, we first perform variable liveness analysis [36] to find the range of statements where each pipelined variable is alive. Next, the FSM state of communication statements and the computation statements are obtained from the scheduling report and the dependency analysis. From the FSM state information of statements, the statement liveness range of each variable is translated into a cycle liveness range. Based on this cycle information, we place a limit on the pipeline stages where each pipelined variable is copied.

For the example in `M2` of `toy_mpath` (Fig. 5), we perform liveness analysis on each variable and find that variable "data" is live in lines 5–6, variable "i" in line 8, and variable "temp" in lines 6–7. Then we assign the states for communication and computation statements in `M2` of `toy_mpath` as was shown in Section V-A1. That is, statements in lines 5, 6, and 8 of Fig. 5 are assigned state 2, and the statement in line 7 is assigned state 6. Based on this information, the statement liveness range is converted into a cycle liveness range—variables "data" and "i" are live at cycle 2 and variable "temp" from cycles 2 to 6. As a result, only variable "temp" is copied through the pipeline stages.

```
01 static bool p1_en[5];                    //enable signal array
02 static int temp[5]; //6             //pipelined variable array
03 static int ptr_st2 = 4, ptr_st6 = 0;   //pipe variable pointers
04 ...
05 else if(M2_state == 2){
06   ...
07   if( p1_en[ptr_st6] == true ){
08     p1_en[ptr_st6] = false;  //disables enable signal after use
09     fifo3_arr[fifo3_wptr++] = temp[ptr_st6]; //7    //read from
10                                         //pipelined variable array
11     fifo3_wnum--;
12   }
13   //conditional blocks for pipeline stages 3, 4, 5 are removed
14
15   if( i_st2 < N ){ //2
16     if( fifo1_rnum != 0 ){ //4
17       p1_en[ptr_st2] = true;    //enables later pipeline stages
18       data_st2 = fifo1_arr[fifo1_rptr++]; //5
19       fifo1_rnum--;
20       temp[ptr_st2] = data_st2*711; //6           //written to
21                                         //pipelined variable array
22       i_st2++; //8
23   } }
24   ptr_st2 = (ptr_st2 + 1) % 5;            // pipelined variable
25   ptr_st6 = (ptr_st6 + 1) % 5;               // pointers update
26 } }
```

Fig. 10. The code after applying pointer-based variable access optimization to the initial code provided in Fig. 8

B. Pointer-Based Variable Access

One of the problems of declaring a pipelined variable for each pipeline stage (as in Fig. 8) is that the same value is copied repeatedly. Assuming a pipelined loop has `I` iterations, `V` variables, and `IL` iteration latency, the complexity of copying pipelined variables is `O(I×V×IL)`.

We propose an alternative method of copying the value of a pipelined variable only once and changing the pointer to the pipelined variable. The modification to the initial code (Fig. 8) is shown in Fig. 10. We first exploit the fact that the value of the pipelined variable is used in the immediate next pipeline stage—thus, the pipeline variable pointer for stage *st* (ptr_{st}) update can be simplified into $(ptr_{st}+1)\%IL$ (lines 24–25). Next, the pipeline variable pointer is shared among all variables and enable signals in the same pipeline stage since all variables and enable signals are copied together to the next pipeline stage if the loop pipeline has not been stalled. An example is shown for the "temp" variable (line 20) and the "p1_en" enable signal (line 17). Note that this optimization has not been applied to variables "i" and "data", because "i" and "data" are only used in pipeline stage 2 (Section VI-A). Finally, we remove the pipeline stage conditional blocks that do not evaluate any statement (line 13—pipeline stages 3, 4, and 5 are removed), because variables and enable signals no longer need to be copied.

The variable access pattern has a similarity with the register rotation technique in IA-64 architecture [37]. Whereas IA-64 used this technique to simplify the register allocation in software pipelining, we use the access pattern to reduce the number of variable copies among the simulated statements.

Since the data is copied only once, the complexity of the pipelined variable copy is `O(I×V)`. The pipelined variable pointers are shared among all variables in the same pipeline

TABLE III
DEBUG DIRECTIVES FOR FLASH

	Syntax	Description	Target
Trigger-related	DEADLOCK	Triggered at deadlock	Module with dataflow pragma
	STALL	Triggered when module or loop has been stalled	Any module or loop
	MODULE_DONE	Triggered when a module completes its execution	Any module
	FIFO_FULL FIFO=<name>	Triggered at FIFO full condition	Any FIFO
	FIFO_EMPTY FIFO=<name>	Triggered at FIFO empty condition	Any FIFO
	EQUAL VAR=<name> VAL=<val>	Triggered when variable equals to value provided	Any stmt with a variable reference
	GREATER VAR=<name> VAL=<..>	Triggered when variable is greater than value provided	Any stmt with a variable reference
Data-related	DUMP VAR=<name> FILE=<name>	Dumps variable data into the specified file	Any stmt with a variable reference
	COMP VAR=<name> FILE=<name>	Triggered when variable differs from golden data in file	Any stmt with a variable reference
Perf-related	TRIP_COUNT	Measures the loop trip count (e.g. for data-dependent loop)	Any loop
	EXEC_CYCLE	Measures the number of execution cycles for module or loop	Any module or loop
	STALL_CYCLE	Measures the number of stalled cycles for module or loop	Any module or loop
	FULL_CYCLE	Measures the number of cycles when FIFO was full	Any FIFO
	EMPTY_CYCLE	Measures the number of cycles when FIFO was empty	Any FIFO

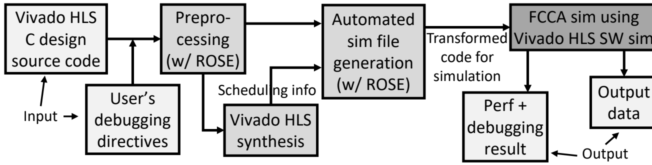


Fig. 11. Overall simulation framework of FLASH

stage—thus, the complexity of the pointer update appears to be $O(I\times IL)$. However, as mentioned in Section V-A1, we group statements to only a few FSM states. The pointer update is not performed on pipeline stages that do not evaluate any statement (e.g., lines 24-25). Assuming there are C stages with communication statements ($C \ll IL$), the pointer update complexity is $O(I\times C)$. Thus, the overall pipeline variable complexity of the proposed method is $O(I\times(V+C))$, which is a large improvement over $O(I\times V\times IL)$.

VII. OVERALL FLOW

The overall simulation framework of FLASH is shown in Fig. 11. Given an input Vivado HLS (VHLS) C design source code, users specify optional debugging directives such as module execution cycle measurement or deadlock triggering (to be explained in Section VIII-B). Then FLASH performs a preprocessing step of adding labels to the source code so that loops and functions can be easily identified. The transformation step uses the APIs in the ROSE [38] and the Merlin [39] compilers. The transformed code is fed into the VHLS for synthesis. Based on the scheduling report given by the HLS tool, the input code is automatically transformed for rapid FCCA simulation (Section V and Section VI). The simulation code has been made compatible with the VHLS software simulator for easy integration with the existing tool. It also allows us to utilize the VHLS's debugging functionality for FLASH's debugging features (details in Section VIII-A). As a final output, FLASH provides the total execution cycles and other user-specified debugging results in addition to the output data that the design is expected to produce.

VIII. SOURCE-LEVEL CORRECTNESS DEBUGGING AND PERFORMANCE DEBUGGING

FLASH provides an option of enabling various source-level correctness and performance debugging features that will be explained in this section.

A. Live Capture

FPGA tools such as Xilinx's ChipScope [40] or Intel's SignalTap [41] capture the data in the FPGA and display it to users for debugging. One of the problems with these configurable logic analyzers is that additional signals often need to be inserted into the capture list to continue tracing the source of a bug after the initial analysis. This requires iterative adjustment of the signal capture list until the bug has been isolated—but the bitstream generation for each analysis often takes hours to finish. Many of the hardware-based HLS debuggers described in Section II also require a long turnaround time due to similar reasons.

Software debuggers, on the other hand, do not require signals to be listed in advance. But due to the lack of cycle accuracy, putting a trigger (breakpoint) on a C source code does not allow the users to observe the signals at a particular cycle of interest. Another problem is that users have limited visibility. For example, local variables in a function different from the trigger cannot be observed unless the user progresses to that function—by which time the content of many variables would have been changed.

To solve these problems, we exploit the fact that FLASH stores the value of all variables (Section V-A5) and the fact that FLASH runs on top of an established commercial tool, VHLS, that provides software simulation debugging features. Upon detection of a trigger condition (details in Section VIII-B), FLASH sets a debug stall flag. All modules are stalled upon the detection of this flag (implementation is similar to the pipelined loop stall modeling in Section V-A4—see step 2 line 1 of Fig. 12). When the simulation has been stalled for debugging, users can step into any function and observe any local variables of interest by adding a new variable into the VHLS debugger's expression window (this is similar to the watch window of Microsoft Visual Studio—see step 4 of Fig. 12). The variables to be captured no longer need to be predetermined.

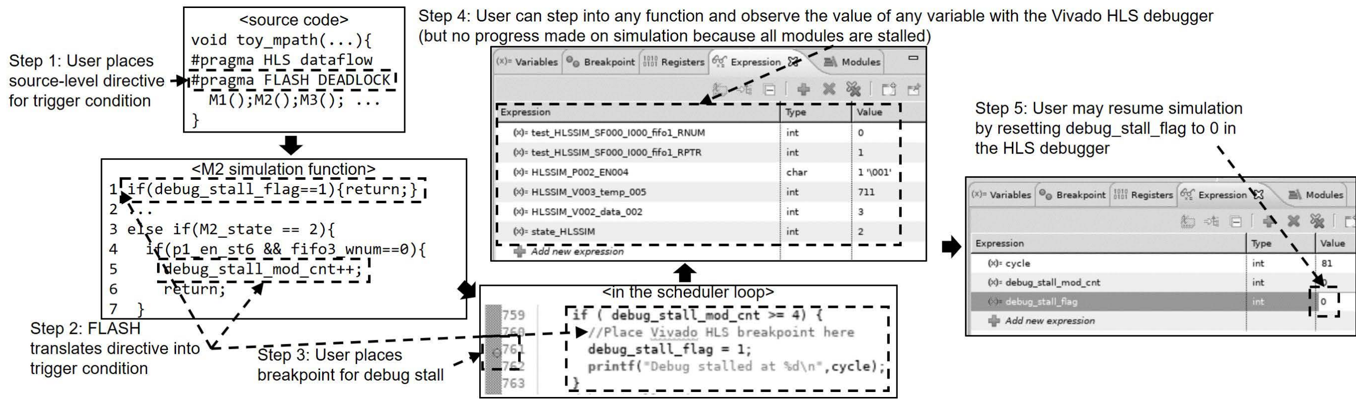


Fig. 12. An example debugging session for deadlock detection using FLASH

The users can expect variables in the FIFO communication and the FSM state variables to match the RTL simulation at all cycles (Section V-A). However, the timing when the variables in the computation statements match the RTL simulation may not be accurate.

In order to pause at the trigger point, FLASH guides the users to place a breakpoint on the simulation code where the debug stall flag is set (step 3 of Fig. 12). The breakpoint is detected by the VHLS debugger. To resume the simulation after observation, the user can modify the value of the debug stall flag to 0 using the expression window (step 5).

B. Source-Level Event Trigger and Performance Measurement

In the Xilinx Chipscope [40], users are required to specify signal names and their value for the tool to start capturing the data (trigger condition). Since the HLS tool applies several transformations in generating RTL file from a C source code, manually identifying the correct trigger condition from an RTL file may be error-prone for novice users. To ease this process, many hardware-based HLS debuggers [11]–[15], [17] allow users to specify variables to be traced or put breakpoints on the source code; however, none of these abstract the trigger condition of events such as deadlock or module/FIFO stall.

FLASH provides a set of source-level directives which can be specified by users to halt computation upon an event of interest. The list is given in the trigger-related row of Table III. The directive is always preceded by: `#pragma FLASH <syntax>`. For example, the deadlock detection directive is `#pragma FLASH DEADLOCK` (step 1 of Fig. 12). FLASH automatically converts a directive into a stall condition that increments a debug variable that counts the number of stalled modules (step 2). For the case of `M2` in Fig. 4, it is stalled if the FIFO is full (line 4 of step 2). If the directive for deadlock detection is found in the source code, FLASH inserts a code that increments the debug variable “debug_stall_mod_cnt” upon stall (line 5 of step 2). After simulating each cycle, FLASH checks to see if “debug_stall_mod_cnt” matches the number of all modules in the design. If so, FLASH sets the debug stall flag that pauses the simulation (Section VIII-A).

FLASH also supports directive-based performance measurement. The list is given in the performance-related row of Table III. The functionality includes module execution and stall cycle measurement, as well as FIFO full and empty cycle measurement.

C. Large Data Debugging

Hardware-based HLS debuggers, such as [11]–[15], [17], optimize the storage and transfer of variable data in an FPGA to be analyzed for correctness. However, the amount of traced data is limited by the BRAM size and the DRAM bandwidth. Being a software-based debugger, FLASH is not limited by the FPGA hardware resource restriction when performing such data-driven debugging—even for multiple variables. Examples include large data dump and large golden reference comparison—the user directives for these functions are shown in the data-related row of Table III.

IX. Experimental Results

A. Experimental Setup

For HLS synthesis, we use the Vivado HLS 2018.2 [2]. For FPGA, we target Xilinx’s Ultrascale KU060 [42]. The target clock frequency is 250MHz. The simulation is conducted with a server node that has an Intel Xeon Processor E5-2680v4 [43] and 64GB of DRAM. The simulation files have been optimized by the Vivado HLS software simulator.

The experiment is performed on `toy_mpath` (Fig. 4) and several dataflow benchmarks: stencil [30], molecular dynamics simulation [18] (Fig. 1), matrix multiplication [44], Cholesky decomposition [45], Needle-Wunsch [46], LU decomposition [47], and sparse matrix-vector multiplication [48]. The benchmarks ([46], [47]) that were not originally designed to execute modules in parallel with FIFO communication have been modified to incorporate this dataflow optimization.

The FLASH simulation result is compared to that of Vivado HLS C and RTL simulation (Verilog), and Verilator 4.012 simulation [29]. Since Vivado HLS RTL files contain core library calls that cannot be processed by Verilator, we have manually replaced them with a behavioral Verilog model.

B. Simulation Time

As mentioned in Section VII, preprocessing, HLS synthesis, and simulation file generation steps are needed to prepare the

TABLE IV
SIMULATION PREPARATION TIME BREAKDOWN

Benchmark	Preproc	HLS Synth	SimFile Gen	Total
Toy_mpath	7.6s	25s	7.9s	40s
Stencil	19s	68s	30s	117s
MD_sim	9.2s	38s	8.8s	56s
Mat_mul	8.7s	36s	12s	57s
Cholesky	15s	98s	37s	150s
NW	18s	99s	26s	143s
LUD	7.1s	21s	8.9s	37s
SpMV	13s	78s	25s	116s

TABLE V
SPEEDUP AFTER APPLYING OPTIMIZATIONS IN SECTION VI (CUMULATIVE SPEEDUP SHOWN)

Benchmark	Avg pipe var depth	Baseline	Var liveness (Section VI-A)	Ptr var acc (Section VI-B)
Toy_mpath	5.4	0.522s (1.00X)	0.483s (1.08X)	0.441s (1.18X)
Stencil	13	2.43s (1.00X)	1.16s (2.09X)	1.12s (2.17X)
MD_sim	34	0.184s (1.00X)	0.0728s (2.53X)	0.0565s (3.26X)
Mat_mul	7.8	0.0802s (1.00X)	0.0722s (1.11X)	0.0716s (1.12X)
Cholesky	8.7	0.0617s (1.00X)	0.0600s (1.03X)	0.0530s (1.16X)
NW	3.2	0.236s (1.00X)	0.224s (1.05X)	0.224s (1.05X)
LUD	19	0.0345s (1.00X)	0.0266s (1.30X)	0.0242s (1.43X)
SpMV	10	0.0853s (1.00X)	0.0808s (1.06X)	0.0803s (1.06X)
AVG	-	(1.00X)	(1.41X)	(1.55X)

files for the proposed simulation. The time breakdown of the steps is presented in Table IV.

The effect of optimizations in Section VI is shown in Table V. The baseline version uses the techniques introduced in our earlier publication [22] and does not have cycle-based variable liveness analysis (Section VI-A) and pointer-based variable access (Section VI-B) optimizations. The table shows that the proposed optimizations result in 1.55X speedup on average. The speedup is greater for benchmarks that have a large (>12) averaged pipeline depth among all variables. The average speedup for Stencil, MD_sim, LUD is 2.28X, and the averaged speedup for the rest of the benchmarks is 1.12X. This is because the proposed optimizations reduce copies of the variables in loop pipelines.

As explained in Section V-A, FLASH uses the FSM state assignment information and the FSM state transition information. The resource allocation / binding information and the component library that exist in RTL code have been abstracted in FLASH, and the computation statements are instead simulated natively on the host machine. The result of this abstraction can be checked in Table VI. FLASH is about 1,630X (=2,800/1.72) faster than the RTL simulation. This confirms our initial speculation that simulating based on the scheduling information greatly accelerates the simulation speed while solving the correctness problems.

Since our flow reflects the scheduling information, we can expect some slowdown compared to the VHLS C simulation. The source of overhead includes the frequent FIFO stalls and

TABLE VI
SIMULATION TIME COMPARISON AMONG VHLS C SIMULATION, VHLS RTL SIMULATION, VERILATOR, AND FLASH SIMULATION

Benchmark	VHLS C Sim	VHLS RTL Sim	Verilator	FLASH
Toy_mpath	0.765s (1.00X)	519s (678X)	120s (157X)	0.441s (0.576X)
Stencil	1.92s (1.00X)	101s (52.6X)	119s (62.0X)	1.12s (0.583X)
MD_sim	0.0652s (1.00X)	89s (1,370X)	7.3s (112X)	0.0565s (0.867X)
Mat_mul	0.0680s (1.00X)	180s (2,650X)	29.2s (429X)	0.0716s (1.05X)
Cholesky	0.0124s (1.00X)	90s (7,260X)	27.7s (2,230X)	0.0530s (4.27X)
NW	0.136s (1.00X)	68s (500X)	27.1s (199X)	0.224s (1.65X)
LUD	0.0319s (1.00X)	129s (4,040X)	16.4s (514X)	0.0242s (0.759X)
SpMV	0.0200s (1.00X)	117s (5,850X)	55.5s (2,780X)	0.0803s (4.0X)
AVG	(1.00X)	(2,800X)	(810X)	(1.72X)

the copy of pipeline variables and enable signals (this overhead was reduced by the optimizations in Section VI as was shown in Table V). However, it is interesting to note in that for some benchmarks such as Toy_mpath and Stencil, FLASH is even faster than the VHLS C simulation (Table VI). This suggests that there is 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 the VHLS C simulator can allocate an unlimited FIFO buffer (Table I). To model FIFO, the VHLS 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 0.765s to 0.128s if we replace FIFO library calls with fixed-size arrays (array size is set to the number of total FIFO elements written). The FLASH simulation flow does not have this problem because the FIFO library calls have been replaced with array-based communication (Section V-A3). The average slowdown of FLASH compared to the VHLS C simulation is 1.72X.

Compared to the RTL simulation, Verilator increases the simulation speed by 3.45X (=2,800X/810X). However, as mentioned in Section II, the speedup is limited because it is difficult to completely remove resource allocation and binding information from the RTL file after they have been added. FLASH does not have this overhead, and as a result, FLASH outperforms Verilator by two orders of magnitude while also achieving the cycle accuracy.

Please note that in our initial research stage, we also evaluated a similar code transformation flow that produces a SystemC simulation file. However, the overhead in the SystemC simulation environment caused a 2-3X slowdown compared to the proposed C-based flow, which motivated us to follow the current approach. Despite the slowdown, SystemC-based approach may be more useful to some tool developers if compatibility with existing SystemC simulation frameworks has a higher priority.

TABLE VII
TOTAL EXECUTION CYCLES ESTIMATED BY VHLS SYNTHESIS REPORT AND FLASH, AND THEIR ERROR RATE COMPARED TO THE RTL-SIMULATED RESULT

Benchmark	RTL sim	Viv HLS syn rpt	FLASH
`Toy_mpath`	4,500,010	4,000,019	4,500,010
	-	(-11%)	(0%)
`Stencil`	524,309	524,299	524,309
	-	(~0%)	(0%)
`MD_sim`	12,089	10,524	12,089
	-	(-13%)	(0%)
`Mat_mul`	330,006	131,075	330,006
	-	(-60%)	(0%)
`Cholesky`	40,741	34,996	40,741
	-	(-14%)	(0%)
`NW`	245,725	131,112	245,725
	-	(-47%)	(0%)
`LUD`	201,260	561,153	201,260
	-	(180%)	(0%)
`SpMV`	163,859	395M	163,859
	-	(240K%)	(0%)

C. Accuracy

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

In Table VII we compare the cycle estimation accuracy with the VHLS synthesis report after we manuallly specify the maximum loop bound in the source code. The estimation error rate is small for `Stencil`, because [30] has a built-in mechanism to allocate adequate buffers to avoid FIFO stalls. For the rest of the benchmarks, we have applied a small (1–2) FIFO depth (e.g., Fig. 4). This causes the FIFO buffer to be frequently full and empty and increases execution cycles. Thus, the HLS synthesis report's estimate is smaller than the RTL simulation result. For `LUD` and `SpMV`, on the other hand, the VHLS tool provides a very large overestimate of the execution cycles. The reason is that these applications have variable loop bounds, and VHLS generates the cycle estimate based on the maximum possible loop bounds [24]. FLASH simulates FIFO stalls and loops with variable bounds in a cycle-accurate fashion, and the estimated execution time accurately matches that of RTL simulation.

X. CONCLUDING REMARKS

With a new HLS software simulation flow based on the scheduling information, we were able to solve the correctness issue and also provide accurate performance estimation. A cycle-accurate simulation result was obtained three orders of magnitude faster than from RTL simulation, because the new simulation flow is not slowed by allocation / binding information and component library. We have described an automated code generation flow that enables this new simulation flow.

We hope that the promising results presented in this work will motivate the HLS commercial tool industry to provide additional routines that simulate 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. Note that in order to increase the readability of the code, we chose to generate the simulation file by transforming it from the source code; but tool vendors may also choose to generate the simulation file from the LLVM IR to exploit the LLVM optimizations.

One limitation of FLASH is that it does not model the stalls from the external memory access. We plan to incorporate this functionality in the future to provide accurate performance estimation for wider range of benchmarks. Another limitation is that FLASH serially simulates the pipelined/task-level parallelism. We plan to parallelize the implementation using Pthread/OpenMP so that large-scale simulation can be performed by exploiting multicore architecture.

Other future work includes allowing FLASH to provide a quick performance estimation for design space exploration of input-dependent benchmarks. Moreover, we hope to incorporate the Intel HLS flow if their tool's synthesis report provides detailed scheduling information in the future.

ACKNOWLEDGMENT

We are grateful to Xilinx for the generous software and hardware donation. We thank Seonmyeong Bak (Georgia Tech.), Professor Miryung Kim (UCLA), Chaosheng Shi (Xilinx), and Professor Zhiru Zhang (Cornell Univ.) for many helpful discussions and suggestions. We would also like to express our gratitude to the anonymous reviewers for their detailed comments and Marci Baun and Janice Wheeler for proofreading this paper.

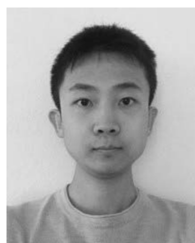
REFERENCES

[1] J. Cong et al., "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, Apr. 2011.
[2] Xilinx. (2018) Vivado High-level Synthesis (UG902). [Online]. Available: https://www.xilinx.com/
[3] Intel. (2019) Intel FPGA SDK for OpenCL Pro Edition. [Online]. Available: https://www.intel.com/
[4] O. Segal et al., "Sparkcl: A unified programming framework for accelerators on heterogeneous clusters," *ArXiv Preprint*, 2015. [Online]. Available: https://arxiv.org/abs/1505.01120
[5] E. Sozzo et al., "A common backend for hardware acceleration on FPGA," in *IEEE Int. Conf. Comput. Design*, 2017, pp. 427–430.
[6] C. Yu et al., "S2FA: an accelerator automation framework for heterogeneous computing in datacenters," in *Proc. Ann. Design Automation Conf.*, 2018.
[7] Mentor Graphics. (2019) ModelSim PE. [Online]. Available: https://www.mentor.com/
[8] Cadence. (2019) Incisive Enterprise Simulator. [Online]. Available: http://www.cadence.com
[9] Synopsys. (2019) VCS Functional Verification Solution. [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html
[10] ——. (2019) ZeBu Fast Emulation. [Online]. Available: https://www.synopsys.com/verification/emulation.html
[11] J. Goeders and S. J. Wilton, "Effective FPGA debug for high-level synthesis generated circuits," in *IEEE Int. Conf. Field Programmable Logic and Appl.*, 2014.
[12] ——, "Using dynamic signal-tracing to debug compiler-optimized HLS circuits on FPGAs," in *IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machines*, 2015, pp. 127–134.
[13] ——, "Signal-tracing techniques for in-system FPGA debugging of high-level synthesis circuits," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 1, pp. 83–96, Jan. 2017.
[14] J. S. Monson and B. L. Hutchings, "New approaches for in-system debug of behaviorally-synthesized FPGA circuits," in *IEEE Int. Conf. Field Programmable Logic and Appl.*, 2014.

[15] ——, "Using source-level transformations to improve high-level synthesis debug and validation on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2015, pp. 5–8.
[16] Y. Choi and J. Cong, "HLScope: High-level performance debugging for FPGA designs," in *IEEE Ann. Int. Symp. Field-Programmable Custom Computing Machines*, 2017, pp. 125–128.
[17] A. Verma et al., "Developing dynamic profiling and debugging support in OpenCL for FPGAs," in *Proc. Ann. Design Automation Conf.*, 2017, pp. 56–61.
[18] J. Cong, Z. Fang, H. Kianinejad, and P. Wei, "Revisiting FPGA acceleration of molecular dynamics simulation with dynamic data flow behavior in high-level synthesis," *ArXiv Preprint*, 2016. [Online]. Available: https://arxiv.org/abs/1611.04474
[19] S. Dai, M. Tan, K. Hao, and Z. Zhang, "Flushing-enabled loop pipelining for high-level synthesis," in *Proc. Ann. Design Automation Conf.*, 2014.
[20] S. Lahti, P. Sjövall, and J. Vanne, "Are we there yet? A study on the state of high-level synthesis," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, May 2019.
[21] P. Coussy et al., "An introduction to high-level synthesis," *IEEE Design & Test of Comput.*, vol. 26, no. 4, pp. 8–17, Jul. 2009.
[22] Y. Chi, Y. Choi, J. Cong, and J. Wang, "Rapid cycle-accurate simulator for high-level synthesis," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2019, pp. 178–183.
[23] A. Canis et al., "From software to accelerators with LegUp high-level synthesis," in *Proc. Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems*, 2013, pp. 18–26.
[24] Y. Choi, P. Zhang, P. Li, and J. Cong, "HLScope+: Fast and accurate performance estimation for FPGA HLS," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 2017, pp. 691–698.
[25] M. Nanjundappa et al., "SCGPSim: A fast SystemC simulator on GPUs," in *Proc. Asia and South Pacific Design Automation Conf.*, 2010, pp. 149–154.
[26] M. Chung, J. Kim, and S. Ryu, "SimParallel: A high performance parallel SystemC simulator using hierarchical multi-threading," in *IEEE Int. Symp. Circuits and Systems*, 2014, pp. 1472–1475.
[27] T. Schmidt, G. Liu, and R. Dömer, "Exploiting thread and data level parallelism for ultimate parallel SystemC simulation," in *Proc. Ann. Design Automation Conf.*, 2017.
[28] A. Mahapatra, Y. Liu, and B. C. Schafer, "Accelerating cycle-accurate system-level simulations through behavioral templates," *Integration*, vol. 62, pp. 282–291, Jun. 2018.
[29] W. Snyder. (2017) Verilator: Speedy Reference Models, Direct from RTL. [Online]. Available: https://www.veripool.org/
[30] Y. Chi, J. Cong, P. Wei, and P. Zhou, "SODA : stencil with optimized dataflow architecture," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 2018.
[31] S. Kundu and I. F. Akyildiz, "Deadlock free buffer allocation in closed queueing networks," *Queueing Systems*, vol. 4, no. 1, pp. 47–56, Mar. 1989.
[32] Y. Choi, "Performance debugging frameworks for high-level synthesis," Ph.D. dissertation, University of California, Los Angeles, Oct. 2019.
[33] Xilinx. (2017) FIFO Generator v13.2 (PG057). [Online]. Available: https://www.xilinx.com/
[34] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: The MIT Press, 2005.
[35] L. Soule and T. Blank, "Parallel logic simulation on general purpose machines," in *Proc. ACM/IEEE Design Automation Conf.*, 1988, pp. 166–171.
[36] R. Allen and K. Kennedy, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA: Morgan Kaufmann, 2002.
[37] Intel. (2000) Intel IA-64 Architecture Software Developer's Manual . [Online]. Available: https://www.intel.com/
[38] ROSE. (2019) ROSE compiler infrastructure. [Online]. Available: http://rosecompiler.org/
[39] Falcon Computing Solutions. (2019) Merlin Compiler. [Online]. Available: https://www.falconcomputing.com/merlin-fpga-compiler/
[40] Xilinx. (2012) ChipScope Pro Software and Cores (UG029). [Online]. Available: https://www.xilinx.com/
[41] Intel. (2019) Quartus Prime Pro Edition Handbook. [Online]. Available: https://www.intel.com/
[42] Xilinx. (2019) UltraScale architecture and product data sheet: overview (DS890). [Online]. Available: https://www.xilinx.com/
[43] Intel. (2016) Intel Xeon Processor E5-2680 v4. [Online]. Available: www.intel.com/
[44] J. Cong and J. Wang, "PolySA: polyhedral-based systolic array auto compilation," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 2018.
[45] J. Liu and J. Cong, "Dataflow systolic array implementations of matrix decomposition using high level synthesis," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2019, pp. 187–187.
[46] B. Reagon et al., "Machsuite: Benchmarks for accelerator design and customized architectures," in *Proc. IEEE Int. Symp. Workload Characterization*, 2014, pp. 110–119.
[47] L. Pouchet. (2015) PolyBench/C. [Online]. Available: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/
[48] Y. Zhang et al., "FPGA vs. GPU for sparse matrix vector multiply," in *IEEE Int. Conf. Field-Programmable Technology*, 2009, pp. 255–262.

Young-kyu Choi (M'10) is a postdoctoral scholar in computer science at the University of California, Los Angeles, where he received his Ph.D. degree in 2019. He received his B.S. and M.S. degrees in electrical engineering from Seoul National University. He developed TV receivers at LG Electronics from 2008 to 2011 and FPGA performance estimation tools at Falcon Computing Solutions in 2017. His current research interests include performance debugging, simulation, and optimization with FPGA high-level synthesis tools.

Yuze Chi received his B.S. degree in electronic engineering from Tsinghua University and started pursuing a Ph.D. degree in computer science in 2016. Yuze's current research interests include software/hardware co-optimization and high-level programming infrastructure for heterogeneous systems.



Jie Wang received his B.S. degree from Tsinghua University, Beijing, China, in 2015. He is currently pursuing the Ph.D. degree at the University of California, Los Angeles. His current research interests include domain-specific architecture design and polyhedral compilation.

Jason Cong (F'00) received his B.S. degree in computer science from Peking University in 1985, his M.S. and Ph. D. degrees in computer science from the University of Illinois at Urbana-Champaign in 1987 and 1990, respectively. Currently, he is a Distinguished Chancellor's Professor at the Computer Science Department, also with joint appointment from the Electrical Engineering Department, of University of California, Los Angeles, the director of Center for Domain-Specific Computing (CDSC), and the director of VLSI Architecture, Synthesis, and Technology (VAST) Laboratory. He served as the chair the UCLA Computer Science Department from 2005 to 2008. Dr. Cong's research interests include novel architectures and compilation for customizable computing, synthesis of VLSI circuits and systems, and highly scalable algorithms. He has close to 500 publications in these areas, including 15 best paper awards, 3 Ten-Year Most Influential Paper Awards, and one inducted to the FPGA and Reconfigurable Computing Hall of Fame. He was elected to an IEEE Fellow in 2000, an ACM Fellow in 2008, and a member of the National Academy of Engineering in 2017.