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 with the designed HLS tool. Experimental results show that FLASH runs three orders of magnitude faster than the RTL simulation.

ACM Reference Format:

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.

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, 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.

<table>
<thead>
<tr>
<th>FIFO depth</th>
<th>Xilinx Viv HLS C Sim</th>
<th>Intel OpenCL HLS Sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlimited</td>
<td>Exact</td>
<td></td>
</tr>
<tr>
<td>Exec model</td>
<td>Sequential</td>
<td>Concurrent</td>
</tr>
<tr>
<td>Feedback</td>
<td>Not supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Sim speed</td>
<td>~5 Mcycle/s</td>
<td>~1 Mcycle/s</td>
</tr>
<tr>
<td>Sim order</td>
<td>Deterministic</td>
<td>Non-deterministic</td>
</tr>
<tr>
<td>Cycle-acc</td>
<td>Not cycle-accurate</td>
<td>Not cycle-accurate</td>
</tr>
</tbody>
</table>
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, 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 require 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 simulation to predict the performance for applications with dynamic behavior. These works, however, do not guarantee cycle-accuracy.

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 PE streams 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 to 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 if FIFO is still empty due to the long latency of M3. Due to the back-pressure from M2 and FIFO, M4 becomes full. Then
we call artificial stall.

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 modules to earlier ones. At the time earlier functions 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.

### 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 executed, the data would not be available. As a result, Vivado HLS simulates the program as if the feedback FIFOs are always empty. Intel HLS compiler avoids the deadlock problem by simulating the feedback data from blocking FIFOs are written in the same cycle. However, the modules deadlock problem by automatically increasing the FIFO depth; this causes 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. The challenge is in transforming C into a form that is 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)
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 corresponds 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 the 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 pipelined 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 FIi.

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

Table 2: Code transformation for FIFO communication

<table>
<thead>
<tr>
<th>HLS source code</th>
<th>Transformed simulation code</th>
</tr>
</thead>
<tbody>
<tr>
<td>fifo.empty()</td>
<td>fifo_rnum == 0</td>
</tr>
<tr>
<td>fifo.full()</td>
<td>fifo_wnum == 0</td>
</tr>
<tr>
<td>data = fifo.read()</td>
<td>data = fifo_arr[fifo_rptr++]; fifo_rnum--;</td>
</tr>
<tr>
<td>fifo.write(data)</td>
<td>fifo_arr[fifo_wptr++] = data; fifo_wnum--;</td>
</tr>
</tbody>
</table>
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).

The overall simulation framework of FLASH is shown in Fig. 10.

The experiment is performed on toy_mopath (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)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Preprocess</th>
<th>HLS Synth</th>
<th>Sim File Gen</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toy_mopath</td>
<td>7.1s</td>
<td>24s</td>
<td>7.3s</td>
<td>39s</td>
</tr>
<tr>
<td>Stencil</td>
<td>15s</td>
<td>60s</td>
<td>22s</td>
<td>97s</td>
</tr>
<tr>
<td>MD_sim</td>
<td>8.0s</td>
<td>35s</td>
<td>11s</td>
<td>54s</td>
</tr>
<tr>
<td>Mat_mul</td>
<td>8.1s</td>
<td>31s</td>
<td>10s</td>
<td>49s</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>V C Sim</th>
<th>V RTL Sim</th>
<th>I OCL Sim</th>
<th>FLASH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toy_mopath</td>
<td>0.602s</td>
<td>492s</td>
<td>4.665s</td>
<td>0.570s</td>
</tr>
<tr>
<td>Stencil</td>
<td>1.46s</td>
<td>113s</td>
<td>7.64X</td>
<td>0.947X</td>
</tr>
<tr>
<td>MD_sim</td>
<td>0.0547s</td>
<td>100s</td>
<td>0.0921s</td>
<td>0.0677s</td>
</tr>
<tr>
<td>Mat_mul</td>
<td>0.0539s</td>
<td>192s</td>
<td>0.201s</td>
<td>0.0810s</td>
</tr>
<tr>
<td>AVG</td>
<td>1.00X</td>
<td>1.570X</td>
<td>3.71X</td>
<td>1.13X</td>
</tr>
</tbody>
</table>

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_mopath 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

### 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 [18] and 64GB of DRAM. The simulation files were compiled with -O3 flag.

#### 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_str & & f_i_f o3 _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, the loop update 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.
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 Stencil1, 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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RTL sim</th>
<th>Vivado HLS</th>
<th>FLASH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toy_mpath</td>
<td>4,500,010</td>
<td>2,000,016</td>
<td>4,500,010</td>
</tr>
<tr>
<td>Stencil</td>
<td>524,309</td>
<td>524,299</td>
<td>524,309</td>
</tr>
<tr>
<td>MD_sim</td>
<td>12,089</td>
<td>10,498</td>
<td>12,089</td>
</tr>
<tr>
<td>Mat_mul</td>
<td>330,006</td>
<td>131,075</td>
<td>330,006</td>
</tr>
<tr>
<td>AVG</td>
<td>-</td>
<td>-13%</td>
<td>0%</td>
</tr>
</tbody>
</table>

8 CONCLUDING REMARKS

By simulating based on the scheduling information, we were able to solve the correctness issue of the software simulaters 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 Phread/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-1717805). We gratefully acknowledge 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