Sphinx: A Hybrid Boolean Processor-FPGA Hardware Emulation System

Ruiyao Pu[1*], Yiwei Sun[1*], Pei-Hsin Ho[3], Fan Yang[1†], Li Shang[2†], Xuan Zeng[1]
[1]State Key Lab of Integrated Chips and Systems, School of Microelectronics, Fudan University, Shanghai, China
[2]China and Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, Shanghai, China
[3]UniVista Industrial Software Group, Shanghai, China

***Abstract*—Existing hardware emulators use either FPGA or Boolean processors, which suffer from long compile time and poor debuggability (FPGA-based), or low emulation performance (Boolean processor-based). This work presents Sphinx, a hybrid Boolean processor-FPGA hardware emulation platform aiming to overcome these shortcomings. Sphinx hardware is a new hybrid architecture that integrates software programmable Boolean processors and FPGAs. Sphinx software is a compilation framework that conducts incremental design partitioning and implements the design-under-test components on Boolean processors and the rest on FPGAs. Together, Sphinx enables an incremental emulation flow and demonstrates high emulation performance, fast compile turnarounds, and good debuggability.**

I. Introduction

Functional verification confirms the logical functionality of digital integrated circuits. With the exponential increase in the complexity and size of integrated circuits, software-based verification methods, such as formal verification and software simulation, have fallen short in terms of scalability and runtime. Hardware emulation, offering orders of magnitude performance improvement over software simulation, has become a must-have functional verification step before silicon tape-out [1].

Existing commercial hardware emulation systems fall into two categories: FPGA-based or processor-based. Cadence Palladium [2] is a Boolean processor-based (BP-based) emulation platform. On the other hand, Synopsys ZeBu [3] and Mentor Veloce [4] are FPGA-based emulation platforms. In addition, the research community has also invested a lot of effort in hardware emulation [5]–[12]. Processor-based emulation platform consists of a massive array of interconnected Boolean processors running at very high speed [1]. It is the compiler's responsibility to partition a design among the processors and produce a schedule of Boolean operations in the correct time order. FPGA-based emulation platform, on the other hand, consists of FPGA arrays with time-multiplexed connections. Compared with processor-based emulation platform, FPGA-based platform has better performance and lower cost. However, since the time consuming placement and routing (P&R) phase is required by FPGA-based emulation, implementing a large design into multiple FPGAs is costly and error prone. During the debugging phase, any changes to fix a design flaw requires re-compilation and P&R, which is time consuming and may require board wiring changes. Furthermore, since FPGA-based emulation platform has limited debugging capability, probing signals inside the FPGAs in real time is difficult and recompiling FPGAs to move probes is time consuming. The comparison between FPGA-based and processor-based functional verification platforms is summarized as follows.

Emulation speed. FPGA-based platform is inherently parallel and offers faster emulation speed than processor-based platform. For Boolean processor-based platform, each emulation cycle typically takes 60-300 clock cycles to complete the emulation of a design. Therefore, even though processor-based platform may be clocked at high speed, its overall emulation speed is slower than that of the FPGA-based platform.

Compilation time. The compilation time of processor-based hardware emulation platforms is shorter than that of FPGA-based hardware emulation platforms because the former only needs to ensure that the Boolean logic operations are in the correct order while the latter spends a lot of time on P&R.

Debuggability. Using the FPGA-based hardware emulation platform, we need to add probes in advance in order to track the signal, with poor signal visibility. While each Boolean processor has its own private data memory for storing intermediate results, thus guaranteeing 100% signal visibility.

In summary, processor-based emulation platform is suitable to support early stage design verification when bugs and fixes are frequent, and FPGA-based emulation platform is suitable to support system prototyping when the design is complete, and high emulation efficiency is required to test lengthy and complex software.

This work aims to exploit the advantages of FPGA-based and Boolean processor-based platforms and avoid their aforementioned shortcomings. The proposed work is based on the observation that in real chip functional verification scenarios, verification engineers usually only debug a small part of the design-under-test, leaving the majority rest unchanged. It is then possible to simultaneously leverage processor to enable good debuggability and FPGA to deliver high emulation speed.

To this end, we propose Sphinx, a new hybrid architecture that integrates software programmable Boolean processors and FPGAs. Key architecture features of Sphinx are summarized as follows. Sphinx consists of hierarchically organized Boolean processor clusters and FPGAs interconnected through dynamically reconfigurable interface circuitry. In Sphinx, Boolean processors are organized in a hierarchical fashion. The Boolean processors within a cluster exchange data via a non-blocking multiplexing array. We designed a hierarchical interconnection network to enable high speed inter-cluster data communication. Furthermore, our research shows that

* Equal contribution
† Corresponding authors: {yangfan, lishang}@fudan.edu.cn.

data exchange instructions account for the majority of the total number of instructions. We design a high-bandwidth multiplexing array to support inter-cluster data communication to reduce the time that Boolean processors wait for external data. Furthermore, we use a reconfigurable interconnect with asynchronous FIFOs to interface between the Boolean processor array and the FPGA. In this way, a Boolean processor can be responsible for processing multiple interface signals, thereby increasing the scalability of the interface circuit.

Sphinx is equipped with an incremental compiler framework to support design partition and compilation between Boolean processors and FPGAs. It conducts incremental partitioning and implements the changed part of design-under-test on Boolean processors and the rest on FPGAs. We pre-store the instructions generated by the compilation framework inside the Boolean processor. Since inter-cluster data communication is significantly more costly than that of intra-cluster data communication, we employ a multilevel k-way partitioning algorithm to cluster the design on Boolean processors so as to reduce the communication overhead with load balancing. Moreover, we design a level-wise scheduling algorithm. Within each level, we try to collapse circuit nodes into the same instruction cycle, and the nodes with greater fan-out have higher priority in order to improve simulation parallelism and reduce processor idle time. In addition, considering that combinational logic occupies the majority of circuits, we simulate nodes while fetching the required data for the next node to be calculated in one combinational logic mapping instruction to further reduce emulation time.

Our contributions are summarized as follows:

- We present a hybrid Boolean processor-FPGA hardware architecture for the first time to leverage the respective advantages of processor-based and FPGA-based hardware emulation approaches for functional verification.
- We present an incremental compiler framework for efficient netlist partitioning and designs on Boolean processor implementation, capable of profiling the flow of all signals in to further depths within the circuits to reduce communication overhead and compilation time.
- We verify our hybrid hardware emulation system with industrial designs. The experimental results demonstrate the significant reduction in compilation time on industrial design cases.

On the Xilinx Virtex UltraScale+ VU19P FPGA, Sphinx achieves up to over 1000x turnaround time optimization compared with FPGA-based emulators, and ensures comprehensive signal visibility with competitive simulation performance between BP-based emulators and FPGA-based emulators.

The remainder of this paper is organized as follows. Section II provides an overview of Sphinx. Section III presents Sphinx hardware architecture. Section IV presents Sphinx software framework. Section V evaluates the performance of Sphinx. This paper is concluded in Section VI.

II. Overview of Sphinx

In this section, we will give a overview of our proposed hybrid BP-FPGA hardware emulation system and the simulation workflow including preprocessing, incremental compilation and hybrid hardware emulation, respectively.

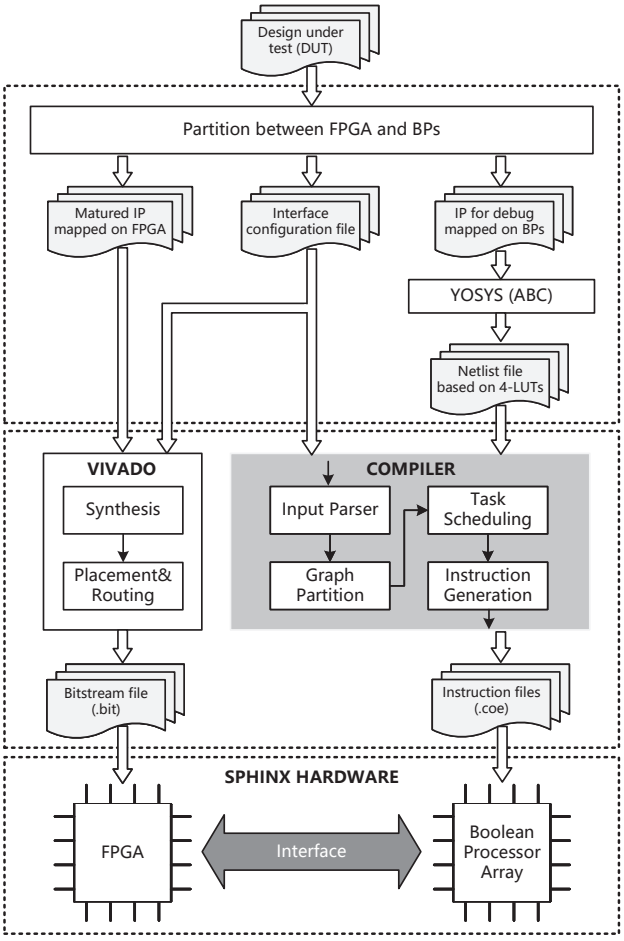

Fig. 1. The flow of our hybrid BP-FPGA hardware emulation system.

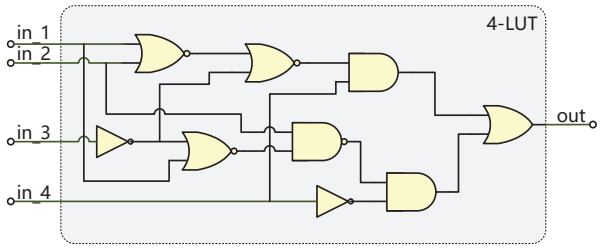

Fig. 2. Example of technology remapping into 4-input LUT.

Fig. 1 illustrates the basic flow of the proposed hybrid hardware emulation system Sphinx. In the preprocessing phase, the design under test (DUT) is read and divided into two partitions: the mature IP mapped on FPGA and the IP for debug mapped on BPs. The latter one generates a netlist file based on 4-input LUT, as shown in Fig. 2, via the open synthesis suite YOSYS [13]. Additionally, an interface configuration file is generated for interface communication, containing the mapping relationships of BP-FPGA reconfigurable bi-directional interface. During the incremental compilation phase, the Sphinx compiler generates instruction files [14] (.coe file) that contain all the instructions for each processor, and Vivado [15] generates a bitstream file [16] (.bit file) that

contains the programming information for the FPGA, which serve as inputs loaded onto the BPs and FPGA to perform hybrid hardware emulation with the reconfigurable interface.

For practical simulation cases, we propose to map part of the circuit that needs frequent modifications stripped from the overall circuit on BPs, and achieve efficient incremental simulation taking advantage of the fast compilation speed and high debuggability of BP-based emulation. It is required that split edges are all FF (flip-flop) edges to ensure the timing coherence of the communication between FPGA and BPs, and the detail partition method will be presented in Section IV-A.

Digital circuits are commonly represented as directed acyclic graph (DAG) [10], in which each node represents a LUT/FF with function, and each directed edge means that one input of the LUT/FF represented by the end node is from the output of the LUT/FF represented by the starting node.

In Sphinx software, we divide the nodes vertically into different clusters based on data dependencies to minimize the communication overhead between clusters, which will be introduced in Section IV-B. Subsequently, the compiler performs task scheduling based on levelization results of the horizontal partitioning. We propose an efficient static scheduling algorithm for the heterogeneous hardware emulation system, and the details will be presented in Section IV-C. Moreover, an instruction generation strategy for our hybrid architecture is proposed in Section IV-D which takes full use of the four-channel data fetching in instructions to hide the time overhead of data waiting.

In Sphinx hardware, we implement a hybrid hardware architecture that allows the size of Boolean processor array to be dynamically scaled, providing flexibility to accommodate DUTs of various sizes. This dynamic scaling helps reduce unnecessary hardware resource overhead, and the details will be provided in Section III-A. Furthermore, Section III-A introduces the innovative instruction set architecture (ISA), where each class of instructions has both computational and data transfer roles. The interface is a crucial component of our hybrid architecture. We propose a dedicated reconfigurable bi-directional interface to ensure proper and efficient data communication, and the details will be introduced in Section III-B.

III. Sphinx Hardware

This section describes the Sphinx hardware architecture which is based on modern FPGA. In our implementation, we use the Xilinx Virtex UltraScale+ VU19P FPGA.

The Sphinx hardware contanins three main components:

- Boolean processor array;
- FPGA;
- Interface: the module responsible for transferring data between Boolean processor array and FPGA.

In the remainder of this section we will introduce the implementation details of Boolean processor array and Interface.

A. Boolean Processor Array

The proposed Boolean processor array consists of a large number of identical bit-wide Boolean processors able to share data with one another through hierarchical interconnection network. These Boolean processors are grouped in clusters, each containing 64 Boolean processors currently. The Boolean processors inside one cluster exchange data through reconfigurable multiplexer arrays, and the clusters are connected to each other through reconfigurable partial crossbar. Each Boolean processor executes the programs generated by the compiler in parallel. All Boolean processors in the array share the same global sequencer (Program Counter) to allow each processor to execute instructions sequentially and synchronously.

TABLE I
Boolean Processor's Instruction Set Architecture (ISA).

Category	Function
Combination Logic Mapping	Emulate combinatorial logic circuit
FF Mapping	Emulate flip-flop
Register Access	Emulate register access operation
Static Configuration	Set the signal values in testbench
Idle	Synchronize to ensure correct timing

The size of the Boolean processor array can be dynamically scaled to match the size of the DUT. This allows for flexible adaptation to various sizes of DUT, thereby reducing unnecessary hardware resource overhead.

1) Instruction Set Architecture (ISA) of Boolean Processor: Boolean processor provides a rich set of instructions to emulate the complex DUT. Its instruction set architecture, shown in TABLE I, has five categories of instructions: Combination Logic Mapping, FF (flip-flop) Mapping, Register Access, Static Configuration and Idle. We describe the functions of each type of instruction in detail below.

The digital circuit can be abstracted into two parts: the combinational logic circuit and the flip-flop. We use two types of instructions, *Combination Logic Mapping* and *FF Mapping*, respectively, to emulate the combinational logic and flip-flop in the DUT and map them to the Boolean processor for computation. The combinatorial logic computation unit in the Boolean processor is a 4-input lookup table (LUT), so one *Combination Logic Mapping* instruction can emulate the combinatorial logic equivalent to a 4-input LUT in the DUT. The Boolean processor generates 1 bit result for each instruction, which is written back to the Boolean processor's private data memory for subsequent use, and the *Register Access* instruction allows us to read the results of previous instructions stored in the Boolean processor's data memory. The function of the *Static Configuration* instruction is to pass the initial excitation value of the testbench into the Boolean processor for subsequent computation. Ideally, we would like all the Boolean processors in the Boolean processor array to execute the instructions without interruption. In practice, however, the input data required for the Boolean processor computation may come from other Boolean processors. Whereas the transfer of data over the interconnect network requires additional Boolean processor clock cycles, we use the *Idle* instruction to keep the Boolean processors in a standby waiting state until the required data arrives. We want to minimize the number of *Idle* instructions by optimizing the Sphinx software compiler.

2) Boolean Processor Unit: Unlike previous processor in processor-based emulator [17], we have designed the novel Boolean processor to handle both computation and data

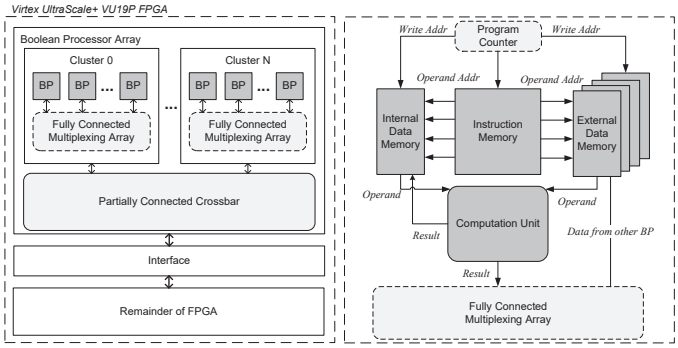

Fig. 3. Sphinx hardware architecture (left) and Boolean processor unit (right).

transfer between processors when executing the four types of instructions, except for the *Idle* instruction. This design enables the Boolean processor to prepare the necessary data for subsequent computations in advance, effectively reducing the waiting time for data. Consequently, this working mechanism significantly enhances the overall performance of our system.

As depicted in Fig. 3, the Boolean processor unit comprises three main components: the instruction memory, internal/external data memory, and computation unit. The program counter, as highlighted within the dashed section, is shared amongst all Boolean processors within the array. The operation of the Boolean processor involves sequential retrieval of prestored instructions from the instruction memory, dictated by the program counter values. These instructions contain fields that serve as indexes to data from the internal/external data memory, thereby emulating the combinational logic/flip-flop within the DUT. The computation results from each instruction are written back into the internal data memory, ready for subsequent calculations. Concurrently, the Boolean processor fetches data from other Boolean processors through the interconnection network, guided by the values in specific fields of the instructions. This data is stored in the external data memory, to be used in future computations.

3) Interconnection Network: The interconnection network within the Boolean processor array is arranged hierarchically and comprises two main parts: intra-cluster and inter-cluster interconnections. Given the higher frequency of data communication among Boolean processors within the same cluster, we utilize a fully interconnected, non-blocking multiplexing array as the internal network within each cluster. Conversely, since data transfer between clusters occurs less frequently, a partially connected crossbar serves as the interconnect network between clusters. This hierarchical structuring of the interconnection network facilitates data transfer between any two Boolean processors within the array, at the advantage of reduced area and clock count overheads. This consequently minimizes the time Boolean processors spend waiting for external data and reduces the number of *Idle* instructions.

B. Interface

The *Interface* module is the key to Sphinx hardware. The architecture of *Interface* is shown in Fig. 4. In summary, it executes the following functions:

- It manages the clock domain crossing data transfers between the Boolean processor array and the FPGA.
- Based on the interface configuration file generated by the Sphinx software compiler, it automatically establishes connections between the corresponding signals of the Boolean processor array and the FPGA.

As mentioned in the overview, we employ Sphinx software to map the parts of the DUT that require debugging as instructions to the Boolean processor array. In contrast, the remaining unchanged parts of the DUT are mapped directly to the FPGA. The connection signals between these two parts of the DUTs is henceforth referred to as the 'Interface Signals'. Both parts operate concurrently, with the interface module facilitating data transmission and synchronization between the Boolean processor array and the FPGA. Our object is to ensure that the operation results of the DUT on the Sphinx hybrid emulation system are the same as those of the DUT directly mapped to the FPGA.

As depicted in Fig. 4, the Interface includes two clock domains: the Boolean processor clock domain and the FPGA clock domain (DUT clock). The FPGA clock period (slow clock) is an integer multiple of the Boolean processor clock (fast clock) period. The multiple relationship between the two is determined by the maximum number of instructions stored in each Boolean processor. Since the maximum operating frequency of the Boolean processor is fixed, the higher the number of instructions, the lower the operating frequency of the FPGA part of the DUT. As mentioned above, the Boolean processor array designed by us is scalable, and the scale of the Boolean processor array can be adjusted adaptively according to the scale of the DUT, so as to improve the running frequency of the FPGA part of the DUT.

The following example, using an FPGA clock cycle (DUT clock cycle), explains how the Interface module functions: After the reset is complete, the Boolean processor array operates simultaneously with the remaining DUT on the FPGA.

From Boolean processor array to FPGA: When all the interface signal calculations are complete, the Boolean processor array sends a flag signal to the interface controller. The interface controller enables the write port of the TXD FIFO. The result of the interface signal calculated by the Boolean processor array is written to the TXD FIFO. When the next rising edge of the FPGA clock arrives, the TXD FIFO outputs the interface signal, connecting it to the corresponding DUT port on the FPGA through a reconfigurable crossbar. The interface configuration file, generated by the Sphinx software compiler, is stored in the configuration SRAM, dynamically configuring the crossbar.

From FPGA to Boolean processor array: Similarly, after the DUT calculations of the FPGA part are complete for all the interface signals, the result data is written into the RXD FIFO. When the Boolean processor array is ready to receive, it sends a flag signal to the interface controller. The interface controller enables the RXD FIFO read port to read data from the FPGA side.

IV. Sphinx Software

Based on our Sphinx incremental compiler as Fig. 1 shows, we present diverse strategies to fit the hybrid architecture and

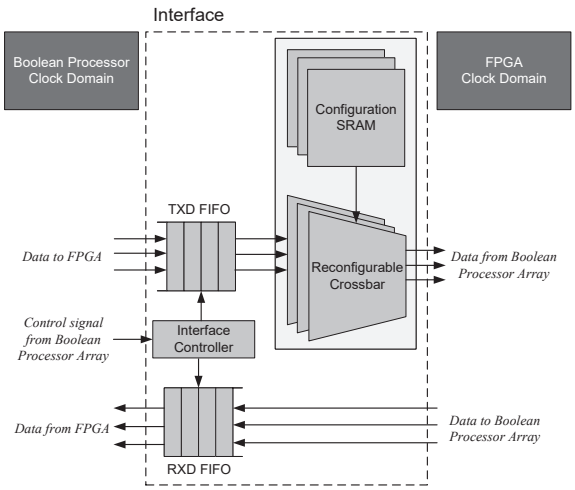

Fig. 4. Reconfigurable bi-directional interface architecture.

boost the compilation performance. Considering the timing problem caused by DUT division, we ensure the partition edges align with flip-flop edges via boundary exploration. To mitigate the higher data transfer latency between cross-clusters, we employ a coarse-grained partitioning for communication overhead reduction. The efficient task scheduling is a key factor to determine the performance of our hybrid hardware emulation system. Finally, the performance is further enhanced through instruction generation using 4-channel fetching, which achieves efficient data propagation.

A. Partition between FPGA and BPs

Since we have to divide the DUT to map on FPGA and BPs separately, the partitioned edges are especially critical to ensure the timing of the interface data transfer.

In sequential circuits, complex functions are usually implemented with flip-flops and combinational logic circuits, where flip-flops are utilized to store and transfer data, and combinational logic circuits are utilized to perform specific logic operations. Considering the clock frequency and delay, the calculation results of the combinational logic circuit can be fully passed to the next flip-flop within one clock cycle. If the circuit divided at the combinational logic edge, the delay from the front or rear flip-flop to the partition edge cannot be predicted, resulting in incorrectness of the interface data transfer. One approach is to sharply reduce the clock frequency to ensure the proper operation of circuits. However, this comes at the expense of severely limiting the simulation performance. Hence, it is crucial to divide the circuit at the flip-flop edge to maintain interface timing consistency and ensure accurate data transfer between the FPGA and BPs, taking into account the frequency and interface transfer delay.

To this end, we determine the combinational logic edges in the input and output interfaces of the divided modules by analyzing the logic functions and connection relationships of the modules, and the nearest flip-flop edge is determined via tracing the circuit connections and clock signal transmission paths starting from the combinational logic division edge. In case of an output interface, the clock signal transmitted from the combinational logic edge to the flip-flop is explorated, conversely, the clock signal transmitted from the flip-flop to the combinational logic edge is explorated. As an result, an interface configuration file is generated, which contains all the partitioned signals at the flip-flop edge.

B. Partition between Clusters

The BP side of our heterogeneous emulation system consists of several clusters, with each cluster containing 64 Boolean processors. The compiler divides the circuits mapped on BPs to different clusters. Data communication between clusters can become the performance bottleneck due to the larger data communication overhead between clusters in comparison to that within a cluster, which requires to minimize the number of cut edges to reduce data waiting. Moreover, workload balance is essential for efficient inter-cluster parallelism, and load imbalance can lead to under-utilization of resources.

METIS [18] is a popular open graph partitioning tool, which provides users with two API interfaces, including the $METIS\_PartGraphKway$ based on k-way algorithm and the $METIS\_PartGraphRecursive$ based on recursive algorithm. We adopt the $METIS\_PartGraphKway$ because the multilevel k-way algorithm is excellent at finding a partition that minimizes the edge cut and balances the computational load across partitions, while the recursive algorithm provided by $METIS\_PartGraphRecursive$ is more suitable for smaller graphs where a strict balance between partitions is not required.

C. Scheduling for Heterogeneous Emulation System

The efficiency of hardware emulation heavily depends on the scheduling of computations. To maximize the benefits of parallelization, it is crucial to allocate and balance workloads across BPs effectively. Moreover, since the heterogeneous features of Sphinx, an improper task scheduling can lead to unsatisfied data transfer timing on the reconfigurable interface resulting in incorrect simulation results. To this end, we propose an efficient scheduling strategy for our hybrid architecture to address these challenges.

There are two broad categories of scheduling, including heuristic algorithms (e.g., list scheduling) and stochastic algorithms (e.g., genetic algorithm). Heuristic scheduling algorithms are more fast and efficient compared to stochastic algorithms, while stochastic algorithms can explore a wide range of possibilities. Considering the requested compilation speed, we employ an optimized list scheduling algorithm to further reduce the compiler runtime while ensuring impressive performance [10].

The scheduling process must conform to the hierarchy of circuits as the simulation is performed level by level, signifying that the simulation of current level will be conducted after that of previous level is completed, consequently, the maximum length of instructions is proportional to the number of levels of the 4-input LUT based netlist.

Levels of the netlist is determined by topological sorting and all nodes in the same level are independent so that they can be simulated in parallel. Fig. 5 illustrates two distinct types of topological sorting algorithms, which provide mobility for

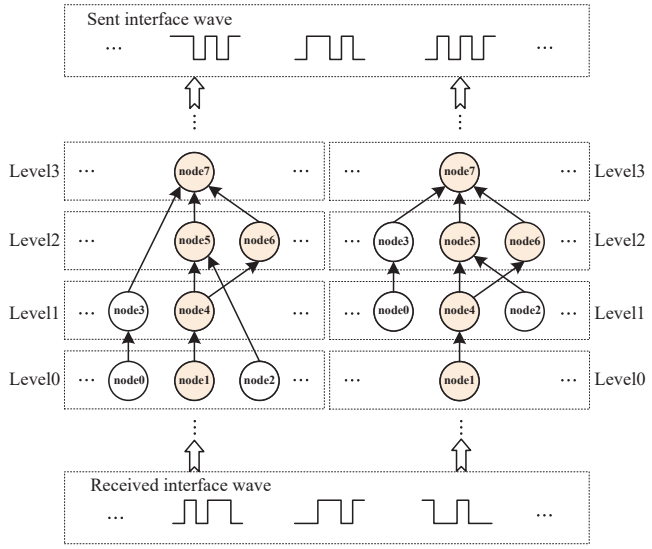

Fig. 5. Types of topological sorting models.

the hierarchy of nodes. On the left, the As-Soon-As-Possible (ASAP) algorithm organizes the nodes in the earliest possible hierarchy. On the right, the As-Last-As-Possible (ALAP) algorithm arranges the nodes in the latest possible hierarchy. It can be seen that the colored nodes in the graph (node 1, 4, 5, 6, 7) have the same hierarchy in both algorithms, which is called critical path nodes, and the number of hierarchies is determined by these critical path nodes. Consequently, these critical path nodes are given precedence when deciding the scheduling priority.

Algorithm 1 illustrates the proposed efficient heuristic scheduling algorithm. Line 1 to line 2 perform ASAP and ALAP algorithm on the DAG topology structure respectively, after which the critical path nodes are identified in line 5 to line 9. Line 10 to line 31 process the scheduling level by level, where all nodes that can be scheduled at the current level are identified, with priority given to critical path nodes and the nodes whose current level is already the latest feasible level. The remaining postponable nodes are then scheduled in descending order of fan-out, ensuring that more nodes receive their input values as early as possible. It is worth noting that the number of levels allocated for postponable nodes cannot exceed the max cycle of scheduling critical path nodes and the nodes with mobility of 0 in each level. This balancing ensures workload distribution and minimizes the number of instructions. The $Allocate\_and\_Collapse$ function is employed to determine the BP number assigned to each node, giving priority to nodes with data dependencies in order to minimize data transfers between BPs and mitigate data conflicts.

D. Instruction Generation

After the task scheduling, we perform instruction generation level by level based on the ISA introduced in Section II-A, and output the .coe file imported into BRAM. For our heterogeneous system, we need to account for both the delay of BPs in receiving data from the FPGA and that in sending data

Algorithm 1 Heuristic Scheduling for Hybrid BP-FPGA Hardware Emulation System

Input: current DAG of circuits on BPs $G = (V, E)$, fanout of each node
Output: 2-D array of nodes for each level $SchList$, allocate results for each node A_{out}

1: $ALAP \leftarrow ALAP\_Schedule(G = (V, E))$;
2: $ASAP \leftarrow ASAP\_Schedule(G = (V, E))$;
3: $N_L \leftarrow MAX(ALAP[v_i])$;
4: $V_{CPN} \leftarrow \phi$;
5: **for** each $v_i \in G$ **do**
6: **if** $ALAP[v_i] - ASAP[v_i] = 0$ **then**
7: $V_{CPN} \leftarrow V_{CPN} + v_i$;
8: **end if**
9: **end for**
10: **for** $L_j = 0$ to N_L **do**
11: $V^{'} \leftarrow \phi$;
12: **for** all $ASAP[v_i] \leq L_j \leq ALAP[v_i]$ **do**
13: $mobility[v_i] \leftarrow ALAP[v_i] - L_j$;
14: $V^{'} \leftarrow V^{'} + v_i$;
15: **end for**
16: sort all v_i in $V^{'}$ by mobility ascending;
17: $max\_cycle \leftarrow 0$;
18: **while** $V^{'} \neq \phi$ **do**
19: **if** $v_i \in V_{CPN}$ || $mobility[v_i] == 0$ **then**
20: $Allocate\_and\_Collapse(v_i, max\_cycle, SchList, A_{out})$;
21: $V^{'} \leftarrow V^{'} - v_i$;
22: **else**
23: sort all v_i in $V^{'}$ by fanout descending;
24: **if** $Allocate\_and\_Collapse(v_i, max\_cycle, SchList, A_{out})$ fail **then**
25: leave v_i for next iteration;
26: **else**
27: $V^{'} \leftarrow V^{'} - v_i$;
28: **end if**
29: **end if**
30: **end while**
31: **end for**

to the FPGA within instructions, aim to ensure the accuracy of transmitted data and the proper operation of the hybrid system. To address this concern, multiple IDLE instructions are included at the beginning and end of the instruction sequence to allow sufficient time for data transmission as shown in Fig. 6. The relationship between instructions and frequency can be expressed as

$$T_{BP} \cdot (N_{receive} + N_{sim} + N_{send}) \leq T_{FPGA}, \quad (1)$$

$$T_{BP} \cdot N_{receive} \geq delay_{receive}, \quad (2)$$

$$T_{BP} \cdot N_{send} \geq delay_{send}, \quad (3)$$

where T_{FPGA} and T_{BP} represent the clock cycle of FPGA and BP, $N_{receive}$, N_{sim} and N_{send} are the number of instructions required for BPs to receive transferred data, perform node emulation and send transferred data respectively, $delay_{receive}$

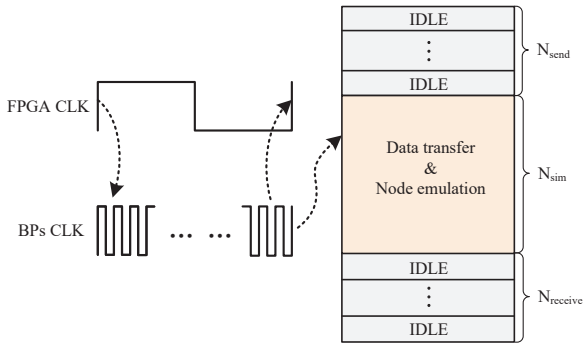

Fig. 6. Instruction model for hybrid BP-FPGA hardware emulation system.

and $delay_{send}$ are the interface delays for BPs to receive and send data.

For minimizing the delay in data availability, we simulate each node while propagating the value of node to all BPs that require it as an input signal. Due to the intricate nature of data dependencies in complex digital circuits, our system supports 4-channel data fetching per instruction, which effectively mitigates potential data congestion. If a data conflict happens, meaning that the desired number of fetches for the current instruction exceeds the available number of channels, the compiler will defer the fetch for the data with a later usage time by one instruction cycle, which ensures efficient handling of data conflicts and prevents the data waiting. In the ideal case with no or few data conflicts, the compiler can allocate only one instruction per node computation.

Algorithm 2 shows the process of instruction generation for BPs. Line 2 to line 4 generate IDLE instructions for BPs to receive transferred data from the FPGA, and instructions are subsequently generated for each node based on scheduling level by level. Line 11 to line 25 perform data propagation based on data dependencies of the simulated node. No data migration between BPs is required in the case that two nodes are allocated to the same BP. If data congestion occurs, the lasted needed node is found and deferred to the nearest instruction cycle with free fetch channels. It is worth noting that if multiple nodes within the same BP require the current simulated node as input, only one data transmission is necessary based on the earliest node. Line 28 to line 30 generate IDLE instructions for BPs to send transferred data to the FPGA, and the judgment condition for the success of instruction generation is shown in line 31.

Algorithm 2 Instruction Generation for Hybrid BP-FPGA Hardware Emulation System Based on 4-Channel Data Fetching

Input: 2-D array of nodes for each level $SchList$, allocate results for each node A_{out}, number of levels N_L, number of instructions required for BPs to receive data $N_{receive}$, number of instructions required for BPs to send data N_{send}, maximum number of instructions while satisfying the timing N_{max}

Output: array of instructions generated for each BP $Instr$

1: array of fetch targets per instruction $FetchList \leftarrow null$;
2: **for** $i = 1$ to $N_{receive}$ **do**
3: $Instr \leftarrow Instr + IDLE$;
4: **end for**
5: **for** $i = 1$ to N_L **do**
6: **for** $j = 1$ to $SchList[i].size()$ **do**
7: $v_i \leftarrow SchList[i][j]$;
8: if not all inputs of v_i are ready then exit;
9: $new\_instr \leftarrow Generate\_Instruction(v_i)$;
10: $Instr \leftarrow Instr + new\_instr$;
11: find all nodes $V^{'}$ fetching the value of v_i as input signal;
12: **while** $V^{'} \neq \phi$ **do**
13: **if** $A_{out}[v_i^{'}] == A_{out}[v_i]$ **then**
14: $V^{'} \leftarrow V^{'} - v_i^{'}$;
15: continue;
16: **else if** $FetchList.size() < 4$ **then**
17: $FetchList \leftarrow FetchList + v_i$;
18: $V^{'} \leftarrow V^{'} - v_i^{'}$;
19: **else**
20: find the node v_{defer} with latest usage time among desired nodes of fetches;
21: find the nearest $FetchList.size() < 4$;
22: $FetchList \leftarrow FetchList + v_{defer}$;
23: $V^{'} \leftarrow V^{'} - v_i^{'}$;
24: **end if**
25: **end while**
26: **end for**
27: **end for**
28: **for** $i = 1$ to N_{send} **do**
29: $Instr \leftarrow Instr + IDLE$;
30: **end for**
31: **if** $Instr.size() \leq N_{max}$ **then**
32: instruction generation success;
33: **end if**

V. Experimental Results

In this section, we perform thorough experiments to demonstrate the achievements of our hybrid BP-FPGA hardware emulation system over three industrial design cases.

We implement the Sphinx hardware on Xilinx Virtex UltraScale+ VU19P FPGA and the Sphinx software runs on a PC with a 3.2GHz AMD Ryzen CPU and 16Gbyte of RAM.

The characteristics of each test case after synthesized using YOSYS are shown in Table II. All three test cases are structured to integrate a RISC-V CPU core named BIRV32 [19] with different encryption modules including DES64 [20], AES128 [21], and SHA256 [22]. The mature RISC-V CPU is unchanged among the three test cases for control and scheduling. While the encryption module is the key of the encryption chip, the engineer needs to modify and test the encryption module frequently to ensure its correct function when designing the encryption chip. Therefore, we can map the encryption module that needs to be frequently modified and debugged in the test cases to the Boolean processor array of Sphinx hardware through Sphinx software, and use Vivado2022.1 to map the unmodified RISC-V CPU in the test cases to the remaining part of Xilinx Virtex UltraScale+

TABLE II
Compilation Time Comparision.

	DUT	Total LUT	Total FF	LUT on BPs	FF on BPs	Compilation Time(s)		Speedup
						FPGA-based	Sphinx	
Case01	BIRV32+SHA256	33776	9506	12002	2787	2000.00	1.32	1515.15×
Case02	BIRV32+AES128	28188	7281	6414	562	1446.00	0.45	3213.33×
Case03	BIRV32+DES64	23068	6900	1294	181	1255.00	0.37	3391.89×

TABLE III
Emulation Performance Comparision.

	DUT	F_{BPmax} (MHz)	N_{BPmax}	F_{Sphinx} $(Mcycles/s)$
Case01	BIRV32+SHA256	135	134	1.01
Case02	BIRV32+AES128	135	55	2.45
Case03	BIRV32+DES64	135	32	4.22

VU19P FPGA. All three test cases originate from open-source projects, thus ensuring easy accessibility for us. Moreover, the design details can be freely discussed without infringing on any intellectual property rights.

A. Emulation speed

The emulation speed of current commercial processor-based emulator like Palladium is 100K∼2M (cycles/s) [1]. While current commercial FPGA-based emulators such as Veloce and Zebu are limited by the number of data transfer pins between FPGAs, with emulation speeds of 500K∼5M (cycles/s) degrading with DUT size [1]. As shown in Table III, our proposed Sphinx hybrid emulation system demonstrates a emulation speed of 1.01M∼4.22M (cycles/s) on three test cases, outperforming processor-based emulation system. Although the performance of our hybrid emulation system Sphinx is slightly worse than that of FPGA-based emulation system, Sphinx has better debug capability compared to FPGA-based emulation system. Unlike using Chipscope [23] or inserting ILA [24], we can take advantage of the fully signal visibility of the Boolean processor array in Sphinx hardware to debug without slowing down emulation speed and without introducing additional logic for debugging.

Furthermore, the emulation speed of Sphinx is determined by

$$F_{Sphinx} = \frac{F_{BPmax}}{N_{BPmax}} \tag{4}$$

where $F_{Sphinx}(cycles/s)$ represents the emulation frequency of Sphinx, $F_{BPmax}(MHz)$ represents the maximum operating frequency of Boolean processor and N_{BPmax} is the maximum number of instructions in a single Boolean processor.

The size of the Boolean processor array in Sphinx is dynamically scalable. So we can scale up the Boolean processor array to reduce the workload of each Boolean processor and decrease the maximum number of instructions N_{BPmax} assigned to a single Boolean processor, thereby further increasing the emulation speed of Sphinx.

B. Compilation time

Table II shows that we obtain the compile speedup from 1515.15× to 3391.89× on Case01-Case03 respectively.

When making changes to the encryption module code, we do not need to re-compile the entire design, we only need to use Sphinx software to re-generate the Boolean processor instructions for the modified encryption module. Thus, we replace the long Vivado P&R time with the short re-generate time of the Boolean processor instructions with Sphinx software, which greatly reduces the debugging iteration of the design.

Actually, there are already simulators that support incremental compilation, while strict limits on modification ratios and the issue of debuggability are still confused. For instance, Vivado's incremental compilation [25] imposes a restriction that the circuit changes should not exceed 5%, and the reduction in compilation time is only around 50%. In contrast, our approach achieves a remarkable speedup of 1500x-3300x.

VI. Conclusion

In this paper, we present Sphinx, a hybrid BP-FPGA hardware emulation system designed for functional verification of digital circuits. Sphinx combines the benefits of two common hardware emulation systems (FPGA-based and processor-based) to achieve enhanced emulation speed, significantly reduced turnaround time and comprehensive signal visibility. The experimental results demonstrate the outstanding performance of Sphinx, with turnaround time surpassing that of FPGA-based emulators 1500x-3300x, and 2× speedup on emulation performance. Furthermore, the improvement in Sphinx compilation speed becomes more pronounced as the percentage of circuits allocated for debugging is reduced. We believe that Sphinx can be employed to diverse designs with different scales and will have remarkable advantages in practical emulation case, especially given the continuous growth in the size of modern circuits.

Acknowledgement

This work is supported partly by National Key R&D Program of China under Grant 2022YFB4400400, partly by National Natural Science Foundation of China (NSFC) research projects 62090025, 62141407 and 61929102, partly by Science and Technology Commission of Shanghai Municipality Project (22DZ1101100).

REFERENCES

[1] L. Scheffer, L. Lavagno, and G. Martin, *EDA for IC system design, verification, and testing*. CRC press, 2018.

[2] Cadence, "Cadence Palladium," accessed Nov. 15, 2022. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html

[3] Synopsys, "Synopsys ZeBu," accessed Nov. 15, 2022. [Online]. Available: https://www.synopsys.com/verification/emulation.html

[4] Mentor Graphics, "Mentor Veloce," accessed Nov. 15, 2022. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/veloce/

[5] K.-S. Oh, S.-Y. Yoon, and S.-I. Chae, "Emulator environment based on an fpga prototyping board," in *Proceedings 11th International Workshop on Rapid System Prototyping. RSP 2000. Shortening the Path from Specification to Prototype (Cat. No.PR00668)*, 2000, pp. 72–77.

[6] Y. Liua, P. Liua, Y. Jiangb, M. Yangb, K. Wua, W. Wanga, and Q. Yaoa, "Building a multi-fpga-based emulation framework to support noc design and verification," *International Journals of Electronics*, vol. 2010, 2010.

[7] J. Ributzka, Y. Hayashi, F. Chen, and G. R. Gao, "Deep: an iterative fpga-based many-core emulation system for chip verification and architecture research," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011, pp. 115–118.

[8] A. Koczor, Ł. Matoga, P. Penkala, and A. Pawlak, "Verification approach based on emulation technology," in *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE, 2016, pp. 1–6.

[9] S. Cadambi, C. S. Mulpuri, and P. N. Ashar, "A fast, inexpensive and scalable hardware acceleration technique for functional simulation," in *Proceedings of the 39th annual Design Automation Conference*, 2002, pp. 570–575.

[10] A. A. Yazdanshenas, "Hardware design and cad for processor-based logic emulation systems." 2006.

[11] M. Kanaan, "A low-cost processor-based logic emulation system using fpgas," 2007.

[12] K. Shi, S. Xu, Y. Diao, D. Boland, and Y. Bao, "Encore: Efficient architecture verification framework with fpga acceleration," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2023, pp. 209–219.

[13] C. Wolf, "Yosys Open Synthesis Suite," accessed 2013. [Online]. Available: https://yosyshq.net/yosys/

[14] AMD, "COEfficient," accessed Nov. 02, 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug896-vivado-ip/COE-File-Examples

[15] XILINX, "Vivado." [Online]. Available: https://www.xilinx.com/developer/products/vivado.html

[16] XILINX, "FPGA Bitstream." [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2018_1/SDK_Doc/SDK_concepts/concept_fpgabitstream.html

[17] W. F. Beausoleil, T.-K. Ng, and H. R. Palmer, "Multiprocessor for hardware emulation," U.S. Patent 5 551 013, Jun. 3, 1994.

[18] G. Karypis, "METIS." [Online]. Available: https://github.com/KarypisLab/METIS

[19] "biRISC-V - 32-bit dual issue RISC-V CPU." [Online]. Available: https://github.com/ultraembedded/biriscv

[20] "DES-Encryption." [Online]. Available: https://github.com/pantao1227/DES_Encryption

[21] "AES-FPGA." [Online]. Available: https://github.com/mematrix/AES-FPGA

[22] "SHA256-Accelerator-Hardware." [Online]. Available: https://github.com/antonson-j1/SHA256-Accelerator-Hardware

[23] XILINX, "Chipscope Pro Software and Cores: User Guide." [Online]. Available: https://china.xilinx.com/content/dam/xilinx/support/documents/sw_manuals_j/xilinx14_7/chipscope_pro_sw_cores_ug029.pdf

[24] XILINX, "Integrated Logic Analyzer v6.2." [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf

[25] XILINX, "Incremental Implementation." [Online]. Available: https://docs.xilinx.com/r/en-US/ug904-vivado-implementation/Preparing-for-Implementation