Fast Switching Activity Estimation for HLS-Produced Dataflow Circuits

Jiantao Liu*, Maksymilian Graczyk*, Andrea Guerrieri†, and Lana Josipović*
*ETH Zurich, Department of Information Technology and Electrical Engineering, Zurich, Switzerland
†EPFL, School of Computer and Communication Sciences, Lausanne, Switzerland & HES-SO, School of Engineering, Sion, Switzerland

***Abstract*—*High-level synthesis* (HLS) tools generate hardware designs from high-level software languages while sidestepping intricate low-level hardware details. However, HLS tools struggle with precise dynamic power estimation and optimization: the high abstraction level they operate on typically contains no or limited information on low-level circuit details that power consumption depends on. *Dataflow circuits* have recently been explored in the HLS context; apart from their ability to achieve performance that is superior to standard HLS-generated circuits, their well-defined structure and computational model offer entirely new opportunities for reasoning about power at the HLS level. This paper exploits this insight to present an accurate switching activity estimator for HLS-produced dataflow circuits. Our estimator combines the knowledge about the dataflow circuit structure with software profiling and detailed glitching analysis to estimate the circuit's switching activity with an average error rate of 1.8% and average speedup of 17.8× compared with a cycle-accurate simulator. Our technology-agnostic solution makes a critical advancement in HLS power estimation and sets the stage for integrating power optimization within the HLS process.**

***Index Terms*—Switching Activity, Power Estimation, High-Level Synthesis, Dataflow Circuits**

I. Introduction

High-level synthesis (HLS) aims to free software developers from the difficulties of detailed hardware design by enabling easier and faster hardware generation from high-level programming languages like C/C++. Yet, despite their reasonable success in balancing the trade-offs between area and performance, HLS compilers exhibit fundamental limitations in accurately estimating and optimizing dynamic power consumption. This stems from the tools' inability to systematically reason about the resultant circuit's structure [39] and, consequently, its switching activity—a fundamental metric for dynamic power estimation. Thus, HLS designers typically resort to measuring the circuit's switching activity after the HLS flow using pre- or post-place-and-route hardware simulation, which can be extremely time-consuming. Furthermore, classic HLS is notoriously inconsistent in the circuit generation process and even the smallest code modification can entirely change the circuit's structure [10]. This variability makes iterative HLS design adjustment based on hardware simulation feedback difficult to converge: while intended to refine the design, such an approach often fails to achieve the desired reduction in power consumption, thus negating the value of this iterative and cumbersome process. Building HLS circuits from predefined blocks could alleviate this issue [39], as it would allow a more predictable and effective power estimation at a high-level design stage. Yet, HLS compilers typically generate customized datapaths and circuit-specific control logic that do not offer such predictability and modularity.

Recent efforts explore the generation of *dataflow circuits* via HLS [5], [12], [13], [20], [26], [37]. These circuits are built out of a small set of predefined and simple dataflow units that use a handshake mechanism to exchange data during circuit runtime. Dataflow circuits have primarily been explored for their performance benefits; yet, their predefined and uniform structure, coupled with the consistency of their communication protocol, holds the potential for mitigating the aforementioned difficulties in HLS switching activity estimation.

Our work seizes this opportunity to build an accurate and fast switching activity estimator for HLS-produced dataflow circuits. We exploit the following insights: (1) The dataflow handshake communication protocol is well-defined and the handshaking rate can be accurately modeled; in Section III, we rely on this information to model the switching activity of the handshake signals. (2) Data exchanges in a dataflow circuit strictly follow the handshake signal exchanges and particular data values are easily obtainable via software profiling; in Section IV, we combine these notions to accurately model the data switching activity of all dataflow units while accounting for possible glitching. The ultimate result, presented in Section V, is an accurate, fast, and technology-independent switching activity estimate obtained immediately during the HLS flow, before producing the final RTL design, which serves as a solid foundation for high-level power estimation and optimizations [7], [27], [39]. On a set of diverse benchmarks obtained from C code, our estimator achieves an average absolute error rate of less than 1.8%; our strategy is, on average, 17.8× faster than obtaining accurate switching values via hardware (i.e., ModelSim) simulation. Our work opens doors for incorporating various power optimizations directly into the HLS flow, without the need for iterative HLS compilations and time-consuming hardware simulations.

II. Background

Recent HLS strategies explore the generation of dataflow circuits from C/C++ code [13], [18]. These circuits are built from a small set of fine-grain *dataflow units* that communicate via *channels* using a consistent and distributed handshake protocol; their runtime scheduling can surpass the performance of standard HLS solutions [20]. An example of a dataflow circuit, implementing the functionality of the code below it, is shown in Figure 1a. Standard operators (i.e., a granularity of a single adder) are connected with dedicated dataflow units [11]: a *merge* receives the initial loop iterator value upon circuit start and the updated iterator otherwise (some implementations

1946-1488/24/$31.00 ©2024 IEEE
DOI 10.1109/FPL64840.2024.00025

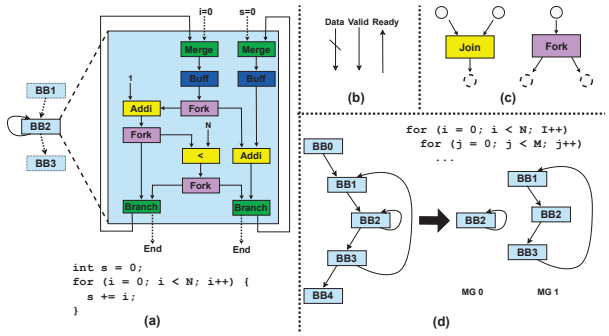


Fig. 1. (a) A dataflow circuit and its CFG obtained from the code below, with an expanded view for the extracted MG (BB2). (b) Dataflow channel. (c) Join and fork dataflow units. (d) CFG (left) and its cycles (right), forming two marked graphs (MGs). BBs are standard compiler basic blocks.

extend a merge into a *mux* with a select input that chooses the operand [13], [20]). A *branch* sends the iterator either to the exit or back to the merge, depending on the loop condition. A *fork* replicates data (i.e., tokens) to send them to multiple successor units. All operators that need multiple input tokens to execute (e.g., add) contain a *join* that synchronizes input tokens. *Buffers* are token-holding elements; they are characterized by their *capacity*, i.e., the number of tokens they can simultaneously store (N_{slots}), and *transparency*, which indicates whether the buffer has a pass-through combinational path from input to output. All dataflow units are connected by dataflow channels as shown in Figure 1b: the valid signal indicates whether the predecessor's output on the channel is valid, and the ready signal indicates whether the successor on the channel is ready to accept the new token. A token transfer occurs when both the valid and ready signals are 1.

Several works modeled dataflow circuit execution and optimized their performance during HLS [6], [22], [35]. The idea is to identify dataflow circuit subgraphs that form *marked graphs* (MGs): choice-free subcircuits that represent program loops, i.e., cycles of the program's control-flow graph (CFG), organized into basic blocks (BBs). All units and the full-line edges of Figure 1a form a marked graph; note that only a single merge input and branch output edge are part of it as, in the repeating MG execution, they consistently issue data only from/to the loop [22]. Decoupling the circuit into regularly-repeating MGs (as in the example of Figure 1d) offers the opportunity to independently reason about their performance: each MG is characterized by its initiation interval (**II**), indicating the number of cycles between two consecutive loop (i.e., MG) executions, and each MG buffer with its *occupancy* $\dot{\theta}$, which reflects how long the buffer holds a data token in MG steady state per **II** clock cycles. Our switching estimator will exploit the notion of MGs and occupancy to systematically analyze steady-state handshake channel switching patterns (Section III) and data channel value propagation (Section IV).

III. Handshake Channel Switching Estimation

Handshake logic significantly contributes to dataflow circuit resources [8], [38]. Thus, accurate handshake switching estimation is important for power estimation. In this section, we first discuss how to estimate the switching activity of handshake signals in the steady state of the MG execution. We then generalize these insights to complete circuit executions.

A. Understanding Handshake Switching in an MG

We here provide our general insights on reasoning about switching in the MG steady state and extend them to concrete unit-specific models in the following sections.

In line with standard pipelined circuits, dataflow circuits exhibit the following property.

Property. *In the steady state of the MG execution, every dataflow channel of the MG completes exactly one token transaction every* **II** *cycles.*

In any pipeline, each register inputs and outputs a value, and each loop computes a value every **II** cycles. The same holds for dataflow circuits: in the steady state of MG execution, every **II** cycles, a single token transaction occurs on every dataflow channel of the MG. Of course, at a specific time, different channels can perform transactions corresponding to computations from different loop iterations, which is exactly the idea and benefit of pipelined dataflow circuits [16].

This property constrains the number and type of feasible switching patterns. For example, it is impossible that a valid switches $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ within an **II**, as it would imply two data transfers per **II** and thus violate the property.

B. How Long Does a Buffer Hold a Token?

Buffers are the elementary storage elements of dataflow circuits; they hold tokens and set the valid/ready signals that accompany them. Thus, buffers regulate the switching of all other nodes by providing the stored signal values to other dataflow units. We therefore use the information on the buffer switching patterns in the MG's steady state to determine the switching of all other units. These patterns are directly determined by how long a buffer holds a token in the steady state: the buffer's valid will be 1 whenever a buffer holds a token, and its ready will be 1 whenever the buffer has sufficient capacity to accept a new token. We can thus use the buffer occupancy $\dot{\theta}$ and capacity N_{slots}, described in Section II, to reason about the valid and ready duration.

The number of clock cycles a buffer holds a data token directly maps to the number of clock cycles the buffer's valid signal stays at 1 (D_v^{buf}):

$$D_v^{buf} = \dot{\theta} \times \mathbf{II}. \tag{1}$$

The buffer occupancy $\dot{\theta}$ indicates the fraction of the **II** that the buffer holds a token; multiplying this value with **II** gives the corresponding cycle count. For example, in Figure 2, Buf_1 has the occupancy of 1/2 in a loop with $\mathbf{II} = 2$; thus, it holds a token and is valid for 1 clock cycle per **II**. This equation holds for a nontransparent buffer—the most intuitive buffer form that corresponds to a classic register. In the rest of the paper, for simplicity, we assume that all buffers are nontransparent; yet, our strategy supports other buffer types [35] as well.

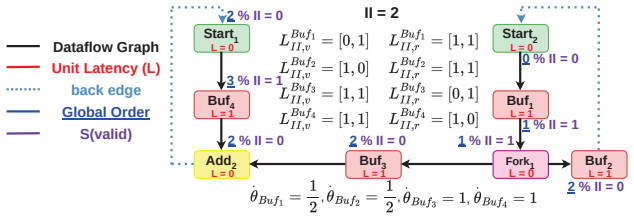


Fig. 2. A dataflow graph annotated with node's unit latency, global order, valid start time, and buffer occupancy. We use these metrics to determine the switching of the handshake signals in the steady-state loop execution.

The status of the buffer's ready signal is determined by how long it is stalled in the steady state, i.e., how long a token resides in its last slot (i.e., the buffer cannot accept a new token and is thus not ready). We can first calculate how many clock cycles the buffer is not ready within the **II** period:

$$D^{buf}_{nr} = \max((\dot{\theta} - (N_{\text{slots}} - 1)) \times \mathbf{II} - 1, 0). \tag{2}$$

The term $\dot{\theta} - (N_{\text{slots}} - 1)$ indicates how long a token is occupying the last slot of the buffer. By multiplying this value with **II**, we get the number of clock cycles in steady state that the buffer is not ready. Then we can get the number of clock cycles when the buffer is ready with $D^{buf}_{r} = \mathbf{II} - D^{buf}_{nr}$.

D^{buf}_{nr} for the single-slot Buf_1 in Figure 2 is equal to 0, which means the buffer is ready for 2 cycles ($D^{buf}_{r} = 2$).

C. When Does a Buffer Hold a Token?

In the previous section, we obtained the *durations* of the buffer's validity and readiness; we also need to know *when* (i.e., in which clock cycles) these signals are set. Others have shown that one can reason about timing relations of memory accesses in dataflow circuits [29]; in this section, we build upon this strategy to determine the start times of the buffer's handshake signals in the MG's steady state.

We define the start time of a signal s of unit n, $\mathcal{S}^n_s$ as the clock cycle when s is (re)set to 1, relative to the start time of the loop iteration. It represents the time when n becomes valid with a new token ($\mathcal{S}^n_v$) or the time it becomes ready to accept a new token ($\mathcal{S}^n_r$). We use the following steps to calculate the valid and ready start times for every buffer in the MG.

We consider a graph of dataflow nodes organized into an MG; each node is characterized by its latency (i.e., sequential delay). We define path $\mathcal{P}(u, v)$ as the longest weighted path of non-repeating nodes from u to v where the weights correspond to the node latency. A cycle $\mathcal{C}(u)$ is the longest weighted path from u to u itself, where all other nodes are non-repeating.

(1) Identify the base node of the MG. To relate nodes in time, we need a common starting point—a clock cycle to consider as the beginning of the **II** interval. We first determine the *throughput-critical cycle* of the MG, i.e., the longest weighted MG cycle whose latency is equal to **II** (in case multiple such cycles exist, we choose any one among them). We define the *base node* as the node that the back edge of the throughput-critical cycle is pointing to. This is the first executed node of this cycle in every loop iteration; its validity indicates the start of the **II** and it thus serves as the origin point to define the timing of all other nodes.

The throughput-critical cycle in Figure 2 is $\mathcal{C}(Start_2) = \{Start_2, Buf_1, Fork_1, Buf_2, Start_2\}$, with a weighted latency of 2. Thus, $Start_2$ is the base node.

(2) Calculate the global order of nodes in the MG. We define the *global order* of node n_i as the latency of the longest weighted path starting from the base node n_b to node n_i:

$$\mathcal{GO}(n_i) = \mathcal{L}_{\mathcal{P}}(n_b, n_i). \tag{3}$$

Here, $\mathcal{L}_{\mathcal{P}}(u, v)$ is the sum of the weights of all nodes in the path. Thus, the global order reflects the latency between the start time of the base node and the node n_i. In Figure 2, global order is the blue underlined number beside each node.

(3) Calculate valid signal start time. The global order specifies the latency between the start of the loop iteration and the validity of a node; yet, our goal is to position the node's validity with respect to the repeating **II**:

$$\mathcal{S}^n_v = \mathcal{GO}(n) \bmod \mathbf{II}. \tag{4}$$

For example, when calculating $\mathcal{S}(Buf_1)$ in Figure 2, the path used to calculate $\mathcal{GO}(Buf_1)$ is: $\mathcal{P}(Start_2, Buf_1) = \{Start_2, Buf_1\}$. Thus, we calculate $\mathcal{S}(Buf_1)$ as $\mathcal{S}(Buf_1) = 1 \bmod 2 = 1$. Equations marked in purple in Figure 2 illustrate the same calculation for the other nodes in the figure.

(4) Calculate the ready signal start time. A buffer's ready is set to 0 only if and when the buffer becomes full (i.e., it can no longer accept new tokens). This can occur only at the time a new token is written into a buffer and the buffer capacity filled, i.e., at the start time of valid. The buffer will become ready again D^{buf}_{nr} (see Equation 2) cycles later:

$$\mathcal{S}^n_r = (\mathcal{S}^n_v + D^{buf}_{nr}) \bmod \mathbf{II}. \tag{5}$$

If the buffer is never full and its ready is never reset (i.e., $D^{buf}_{nr} = 0$), the buffer is always ready; a new iteration of ready starts at the same time the buffer becomes valid, as the equation above suggests.

(5) Calculate active ranges. We define a node n's *active range*, $L^n_{II,s}$, as a list of **II** elements representing the value of s in each clock cycle in steady state. For example, $L^{buf}_{II,v} = [1, 0]$ means the buffer's valid signal is 1 in the clock cycle 0 and 0 in clock cycle 1 in the MG steady state.

We define the set of clock cycles when the value of a signal s of node n is 1 according to a specific signal's start time ($\mathcal{S}^n_j$) and the duration (D^n_s) of signal s as follows:

$$\mathcal{A}(\mathcal{S}^n_s, D^n_s) = \{\mathcal{S}^n_s, ..., \mathcal{S}^n_s + D^n_s - 1\} \bmod \mathbf{II}. \tag{6}$$

We express the active range of signal s of node n as:

$$L^n_{II,s}[i] = \begin{cases} 1, \text{if } i \in \mathcal{A}(\mathcal{S}^n_s, D^n_s) \\ 0, \text{else} \end{cases}, \forall i \in [0, \mathbf{II}). \tag{7}$$

Then a buffer's valid active range ($L^{buf}_{II,v}$) can be calculated based on $\mathcal{A}(\mathcal{S}^{buf}_v, D^{buf}_v)$. Similarly, the buffer's ready active range ($L^{buf}_{II,r}$) can be calculated based on $\mathcal{A}(\mathcal{S}^{buf}_r, D^{buf}_r)$.

In Figure 2, both active ranges for all buffers are shown in the middle of the figure and are calculated based on the buffer occupancy information shown at the bottom.

Our analysis assumes fixed operator latencies; in a dataflow system, this is not always the case (e.g., memory accesses can have variable latencies [17]). In line with prior dataflow circuit modeling strategies [21], [29], we assume latencies that provide the best circuit performance; yet, our strategy is general and supports any latency-determining scheme.

D. Handshake Properties of Other Dataflow Units

Buffers hold data tokens and regulate the handshake channel switching activity by dictating the rates with which tokens are exchanged. All units between buffers simply propagate the valid and ready signals following unit-specific rules (i.e., their logic functions). We here detail these rules for the join and fork. As an MG has no conditional execution (see Section II), the rules for its other units directly derive from these units.

Join Node: A join is valid in the clock cycles when all of its predecessors are valid (see Section II and Figure 1b). Therefore, the join's valid active range can be calculated as:

$$L_{II,v}^{join} = L_{II,v}^{pred_1} \,\&\, \ldots \,\&\, L_{II,v}^{pred_n}, \forall pred_n \in U. \tag{8}$$

Here, U is the set of all join's predecessors, $L_{II,v}^{pred_n}$ is the valid signal active range for the corresponding predecessors, and $\&$ is a logic *and* that operates on pairs of individual elements of the lists. For example, the valid active range of Add_2 in Figure 2 can be calculated as $L_{II,v}^{Add_2} = L_{II,v}^{Buf_4} \,\&\, L_{II,v}^{Buf_3} = [1,1]$.

For a join's input channel to be ready, its successor needs to be ready and all other inputs must be valid [15]. Thus, we model the ready active range of join's input channel i as:

$$L_{II,r_i}^{join} = L_{II,v}^{pred_1} \,\&\, \ldots \,\&\, L_{II,v}^{pred_j} \,\&\, L_{II,r}^{succ}, \tag{9}$$

where $L_{II,v}^{pred_j}$ is the valid active range of the input channel other than input channel i.

Fork Node: A fork is ready when all of its successors are ready (see Section II and Figure 1b). Thus, we calculate its ready active range as:

$$L_{II,r}^{fork} = L_{II,r}^{succ_1} \,\&\, \ldots \,\&\, L_{II,r}^{succ_n}, \forall succ_n \in E, \tag{10}$$

where E is the set of all fork's successors, and $L_{II,r}^{succ_n}$ is the ready active range for the corresponding successor.

The fork's valid is set from the clock cycle its predecessor's valid is set (i.e., the predecessor is sending a token) until the end of the clock cycle in which the corresponding successor's ready is set (i.e., the successor has accepted the token) [11]. This pattern repeats every **II** cycles and determines whether the fork's valid switches. We first calculate the number of clock cycles between the start time of the fork's predecessor's valid and the j-th successor's ready as follows:

$$D_{v_j}^{fork} = \mathcal{S}_v^{pred} - \mathcal{S}_r^{succ,j} + \mathbf{II} - 1. \tag{11}$$

Here, $\mathcal{S}_s^n$ is the start time of signal s of node n. Based on Equation 6 and 7, the valid active range (L_{II,v_j}^{fork}) of channel j can be calculated based on $\mathcal{A}(\mathcal{S}_{v_j}^{fork}, D_{v_j}^{fork})$, where $\mathcal{S}_{v_j}^{fork} = \mathcal{S}_v^{pred}$. Consider the channel between $Fork_1$ and Buf_3 in Figure 2; $D_{v_1}^{Fork_1} = \mathcal{S}_v^{Buf_1} - \mathcal{S}_r^{Buf_3} + 2 - 1 = 1$, thus, $L_{II,v_{Buf_3}}^{Fork_1} = [0,1]$.

E. Handshake Signal Switching Estimation

We developed an event-driven algorithm that calculates all steady-state timing and switching properties for the buffers in the MG, as discussed in Section III-C. It then iteratively updates all other nodes based on the models of Section III-D, until all active ranges and signal start times have been determined. During dataflow circuit construction, at least one buffer is inserted into every combinational cycle to ensure functional correctness [22]; thus, there are no combinational cycles between any of the valid and ready handshake signals, and no cyclic dependencies between their active ranges. Therefore, the algorithm is guaranteed to terminate with the handshake signal active ranges for all nodes in the MG determined.

The result for Figure 2 is: $L_{II,v_{Buf_3}}^{Fork_1} = [0,1]$, $L_{II,v_{Buf_2}}^{Fork_1} = [1,1]$, $L_{II,r_{Buf_1}}^{Fork_1} = [0,1]$, $L_{II,r_{Buf_3}}^{Add_2} = [1,1]$ and $L_{II,r_{Buf_4}}^{Add_2} = [1,1]$.

To determine the number of switches for each signal s per **II**, $N_{II,s}^n$, we count the number of bit changes across its active range $L_{II,s}$ by performing a bitwise *xor* of all neighboring list elements as well as the first and the last one (to account for the switching on the iteration transitions). For example, in Figure 2, $N_{II,v_{Buf_3}}^{Fork_1} = 2$. We thus obtained the steady-state switching patterns for all nodes in an MG of a dataflow circuit.

F. From MG to Complete Program Execution

We now generalize our switching estimation from a single MG (i.e. a single program loop) to an entire program.

We leverage the fact that our dataflow circuits are produced by HLS from sequential C code, which contains straight pieces of code (i.e., nonrepeating elements) and repeating loops (i.e., MGs); if we can determine the sequence of these code sections, we can compose their switching profiles accordingly to obtain the switching profile of the entire program. We obtain this information via IR-level software profiling: (1) We collect the BB execution trace and use it to construct a sequence of straight code segments and repeating MGs. (2) In every straight (i.e., nonrepeating) code segment, each handshake signal switches twice (e.g., a unit becomes valid once and is never valid again); for all MGs in the sequence, we use the procedure of the previous section to determine the steady state node switching and multiply it by the number of MG executions. (3) A single node can belong to several MGs; we thus sum up the switching of each node across the different MG executions. This provides us with the complete switching activity profile of all nodes during the entire program execution.

IV. Data Channel Switching Estimation

In this section, we combine the information from IR-level software profiling and the timing relations discussed above to estimate the switching activity of data channels.

(1) Obtain buffer values via software profiling. Identify the value producer for each buffer b (i.e., the operator whose value is written into b). Use software profiling to obtain a list of values computed by the producer and written into b, $L_{profile}^b$.

(2) Determine buffer outputs over time. In the loop steady state, each value of $L_{profile}^b$ will be held by buffer b and observable at its output for **II** cycles. Create list L_{time}^b

which replicates each element of $L^b_{profile}$ **II** times and, thus, reflects the time interval in which each value is held in b:

$$L^b_{time}[\mathbf{II} \times i, \mathbf{II} \times (i+1) - 1] = L^b_{profile}[i], \forall i \in N, \quad (12)$$

where N is the number of loop iterations (and, thus the number of elements in $L^b_{profile}$).

(3) Align buffer lists to account for sequential delays. Different buffers produce and consume values at different times, based on the sequential delays between them. Determine list L^b_{final} where $L^b_{final}[c]$ represents the buffer's output value in clock cycle c by shifting L^b_{time} of each buffer b for $\mathcal{S}^b_v$:

$$L^b_{final} = L^b_{time} \gg \mathcal{S}^b_v, \quad (13)$$

where $\mathcal{S}^b_v$ is the start time of b. The buffer's output value is 0 during the first $\mathcal{S}^b_v$ clock cycles: $L^b_{final}[0, \mathcal{S}^b_v - 1] = 0$.

(4) Determine combinational node values. Buffers set the values that are, on every clock cycle, combinationally propagated through other (i.e., combinational) operations. Starting from the input buffers, traverse all combinational subgraphs of the loop (i.e., combinational regions delimited by input and output buffers) and assign output values to each node based on the node's function $\mathcal{F}_{op}$ and its predecessor values:

$$L^{op}_{final}[c] = \mathcal{F}_{op}(L^{p_1}_{final}[c], ..., L^{p_n}_{final}[c]), \forall c \in Cycles. \quad (14)$$

(5) Calculate switching of each node. Iterate through L^{op}_{final} of each node and calculate the switching between consecutive list values (i.e., the Hamming distance of the binary value representations). Sum up the switching to obtain the total switch count of *op* during a loop execution.

(6) Extend to entire program execution. A program execution is a sequence of executions of consecutive loops and straight datapaths. Repeat steps 1-5 above for each loop. Straight datapaths are treated analogously; the node values are determined by the preceding buffers and operators.

This strategy reflects the shifts in operator production and consumption and, thus, captures situations where operand values misalign and operator results glitch. For example, in Figure 2, Add_2 receives inputs from Buf_4 and Buf_3. Based on the discussion above, $L^{Buf_4}_{final} = L^{Buf_4}_{time} \gg 1$, and $L^{Buf_3}_{final} = L^{Buf_3}_{time} \gg 0$, which means that Buf_3 produces an operand for the adder one clock cycle before Buf_4 produces the corresponding operand. During this cycle, the adder will glitch as it will calculate the addition of operands from different loop iterations. Similarly, our strategy accounts for data glitches at places where control flow points meet (i.e., glitching of multiplexers at BB inputs [20], see Section II).

V. Evaluation

In this section, we evaluate the effectiveness of our switching activity estimator.

A. Methodology and Benchmarks

We implement our open-source switching estimator (github.com/EPFL-LAP/dynamatic) in Dynamatic [19], an open-source HLS compiler based on MLIR [24], that generates dataflow circuits from C/C++ code. We implement the software profiler (Section IV) in MLIR's structured control flow (SCF) IR to extract operation values and BB execution traces. We use the dataflow graph generated by Dynamatic, already annotated with unit latencies and buffer occupancies, to calculate steady-state handshake switching (Section III), and the data switching and glitches (Section IV).

We evaluate benchmarks from standard HLS suites [34] and recent works [8], [18], with different execution patterns and control flow structures. They contain up to 373 dataflow units and up to 7 MGs; the largest is *gemver* with 7589 LUTs. In all benchmarks, the total number of loop iterations is above 10,000; thus, the loop steady state executions represent the majority of the circuit's runtime. We synthesize all benchmarks targeting a clock period of 4 ns. All test inputs are randomly generated. We compare our estimation with switching values obtained from VCD files produced by ModelSim SE 10.7b simulations, together with an in-house switching analyzer; if a signal has multiple toggles within one clock cycle caused by delta cycles in ModelSim simulation, we only take the final status in the analyzer. We run all experiments on a computer with AMD Ryzen 7 Pro 7840U CPU and 32 GB RAM.

B. Results: Accuracy of Total Switching Count Estimation

In Figure 3a, we evaluate the accuracy of our estimator in determining the total switching count (i.e., throughout the entire circuit execution) of dataflow channels (i.e., data and handshake signals exchanged between dataflow units) by comparing the results of our estimator with those obtained from ModelSim simulation. We use *activity ratio error* [23] as our metric. It divides the sum of the estimated switching by the sum of the simulated number of switches:

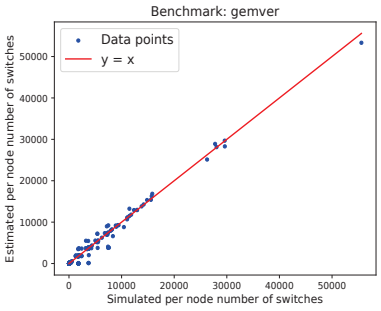
$$a.r.e. = \frac{\sum_{n \in \text{circuit}} N_{\text{est}}(n)}{\sum_{n \in \text{circuit}} N_{\text{sim}}(n)} - 1. \quad (15)$$

Data channels. We compare two data channel switching estimation strategies with the ModelSim reference: a naive software profiling approach (indicated as w/o glitches in the table) and the approach of Section IV (w glitches) that accounts for timing relations between nodes and, thus, captures data glitching. The former exhibits a notable error with respect to the ModelSim baseline (i.e., up to 33.1%) due to its inability to account for data glitching. In contrast, our strategy of Section IV successfully captures both actual data exchanges and data glitches, resulting in an average absolute error of only 1.83%. These outcomes point to the significant contribution of glitches to the circuit's switching and the effectiveness of our data switching estimation in accounting for this effect.

Handshake channels. We evaluate the accuracy of our handshake switching estimation from Section III. Note that software profiling is inapplicable here as it has no information on the circuit's topology and its handshake logic. As shown in Figure 3a, our estimator exhibits average absolute errors of 1.13% and 4.28% for the valid and ready signal, respectively. Interestingly, the errors are higher in benchmarks with irregular control flow (e.g., *gcd*, *gsum*) than those with regular control flow (e.g., *simple* and *fir*), where our estimation is

Benchmark	Data Channel (%) a.r.e. (w/o glitches)	Data Channel (%) a.r.e. (w glitches)	Handshake (%) a.r.e._v	Handshake (%) a.r.e._r	Overall (%) a.r.e.
simple	-12.5	0.3	0.0	0.0	0.0
fir	-24.8	0.0	0.0	0.0	0.0
gcd	-8.6	-8.6	2.3	-7.5	-5.7
matvec	-33.1	-0.1	0.0	0.5	-0.1
bicg	-29.5	-0.5	0.4	5.4	0.8
matrix	-20.2	-1.8	1.2	-2.7	-0.7
gemver	-27.4	-2.1	-2.7	-7.5	-2.5
gsum	-11.5	1.2	2.4	10.6	4.6
Avg. |a.r.e.|	20.95	**1.83**	1.13	4.28	**1.8**

(a)



(b)

Benchmark	Modelsim Runtime (s)	Estimator Runtime (s)	Speedup
simple	11.24	3.70	3.0×
fir	46.42	1.43	32.5×
gcd	2.19	0.64	3.4×
matvec	46.10	2.80	16.5×
bicg	107.66	3.98	27.0×
matrix	197.00	21.70	9.1×
gemver	37.34	1.98	18.9×
gsum	68.96	2.17	31.8×
Average	61.61	4.80	**17.8×**

(c)

Fig. 3. (a) Activity ratio error for data and handshake signals. The data channel a.r.e. (w glitches) represents our solution. (b) Per-node estimation switches versus simulated switched for benchmark *gemver*. Our per-node switching error is consistently low for all signals. (c) Runtime comparisons between our estimator and ModelSim. Our approach is significantly faster across all benchmarks.

error-free). This is due to the reduced accuracy of Dynamatic's occupancy calculator in such benchmarks and orthogonal to our contribution—an accurate switching activity estimator. The systematically low error rates across different signals and their combined low error in the final column of the table (**Overall**) show that we successfully achieved this goal.

In Figure 3b, we plot the per-node estimation error for our largest benchmark, *gemver*. The blue points represent the relation between the estimated ($N_{\text{est}}(n)$, y axis) and simulated ($N_{\text{sim}}(n)$, x axis) switches for each signal of the circuit; the distance of each point from the red line is proportional to its estimation error. The proximity of the blue points to the red line indicates that we successfully achieved a low per-node estimation error. Other benchmarks follow the same trends.

C. Results: Runtime Comparisons with ModelSim

In this section, we evaluate the runtime benefits of our simulator over ModelSim simulation. We only record dataflow unit ports in the first-level design hierarchy in ModelSim simulations and omit internal unit structures; this aligns with the information obtained by our estimator. Figure 3c shows our runtime comparison. The main takeaways are: (1) The runtime of ModelSim significantly differs across benchmarks, as it is primarily dictated by the circuit's execution runtime (i.e., the number of cycles to simulate) and complexity (i.e., the number of waveforms to generate). Our simulator's runtime is significantly less variable. The only notable exception is *matrix*, where the dominant component to our code's runtime is the cycle enumeration algorithm for back edge identification (see Section IV); improving our relatively unoptimized code implementation could further reduce this variability. (2) Our estimator is significantly faster than ModelSim simulation and achieves speedups of up to 32.5×. This, coupled with the accuracy of our approach, points to its effectiveness in high-level switching activity estimation and the feasibility of using our strategy for early-stage power estimation in HLS.

VI. Related Work

Previous works use machine-learning-based [25], [27], [28] and model-based approaches [14], [31], [40] to estimate dynamic power at a high abstraction level. They primarily rely on unit switching activities derived from timing-agnostic, IR-level simulations; they do not consider glitches and cannot model dataflow handshake signal switching. Many are application-specific, whereas our strategy is general. Several studies [4], [32], [33] used stochastic models to approximate switching activities, typically assuming a zero-delay model [2]; this is in line with the subpar w/o glitches results of Figure 3a. Several works [3], [23] introduced simulation-centric methods for glitch prediction; they start from a post-synthesis netlist, whereas our estimator works at a higher abstraction level, runs faster, and provides opportunities for high-level power optimization [39]. Fast cycle-accurate simulators [1], [9], [30], [36] can capture the behavior of the circuits generated by HLS tools. Yet, just like standard hardware simulation, these approaches are highly dependent on the number of clock cycles to simulate; in contrast to our estimator, their high abstraction level does not contain sufficient structural information about the circuit to capture all switching and glitches.

VII. Conclusion

Although effective in exploring various area and performance metrics, HLS significantly lags in power estimation and optimization. This is due to the inability of classic HLS compilers to systematically reason about the circuit's structure and switching activity, a prerequisite for dynamic power estimation. HLS of dataflow circuits offers a unique opportunity to reason about power at HLS level: As these circuits are built out of a small set of well-defined blocks systematically composed by the HLS compiler, the circuit's switching activity can be accurately and quickly computed. This paper validates this insight: we present a fast and accurate switching activity estimator for dataflow circuits obtained from C code that produces almost identical switching results as those obtained via hardware simulation, while speeding up the process for up to 32.5×. The effectiveness of our estimator opens doors to fast and power-aware HLS design and optimization.

VIII. Acknowledgement

This work has been supported by the Swiss National Science Foundation (grant number 215747).

REFERENCES

[1] M. Abderehman, J. Patidar, J. Oza, Y. Nigam, T. A. Khader, and C. Karfa. FastSim: A Fast Simulation Framework for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(5):1371–1385, May 2021.

[2] J. H. Anderson. *Power Optimization and Prediction Techniques for FPGAs*. PhD thesis, University of Toronto, 2005.

[3] J. H. Anderson and F. N. Najm. Switching Activity Analysis and Pre-Layout Activity Prediction for FPGAs. In *Proceedings of the 2003 International Workshop on System-Level Interconnect Prediction*, page 15–21, New York, NY, USA, Apr. 2003.

[4] A. Bogliolo, L. Benini, B. Riccó, and G. De Micheli. Efficient Switching Activity Computation during High-Level Synthesis of Control-Dominated Designs. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, pages 127–132, San Diego, CA, USA, Aug. 1999.

[5] M. Budiu, P. V. Artigas, and S. C. Goldstein. Dataflow: A Complement to Superscalar. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 177–86, Austin, TX, Mar. 2005.

[6] D. Bufistov, J. Cortadella, M. Kishinevsky, and S. Sapatnekar. A general model for performance optimization of sequential systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 362–69, San Jose, CA, Nov. 2007.

[7] D. Chen, J. Cong, Y. Fan, and Z. Zhang. High-Level Power Estimation and Low-Power Design Space Exploration for FPGAs. In *Proceeding of the 12th Asia and South Pacific Design Automation Conference*, pages 529–534, Yokohama, Japan, Jan. 2007.

[8] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson. Combining Dynamic & Static Scheduling in High-Level Synthesis. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 288–98, Seaside, CA, Feb. 2020.

[9] Y.-K. Choi, Y. Chi, J. Wang, and J. Cong. Flash: Fast, Parallel, and Accurate Simulator for HLS. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4828–4841, Jan. 2020.

[10] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Transactions on Reconfigurable Technology and Systems*, 15(4):1–42, Aug. 2022.

[11] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of Synchronous Elastic Architectures. In *Proceedings of the 43rd Design Automation Conference*, pages 657–62, San Francisco, CA, July 2006.

[12] S. A. Edwards, R. Townsend, and M. A. Kim. Compositional Dataflow Circuits. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 175–84, Vienna, Sept. 2017.

[13] A. Elakhras, A. Guerrieri, L. Josipović, and P. Ienne. Unleashing Parallelism in Elastic Circuits with Faster Token Delivery. In *Proceedings of the 22nd International Conference on Field-Programmable Logic and Applications*, pages 253–61, Belfast, UK, Aug. 2022.

[14] T. Jiang, X. Tang, and P. Banerjee. Macro-models for High Level Area and Power Estimation on FPGAs. In *Proceedings of the 14th ACM Great Lakes Symposium on VLSI*, page 162–165, New York, NY, USA, Apr. 2004.

[15] L. Josipović. *High-level Synthesis of Dynamically Scheduled Circuits*. PhD thesis, EPFL, 2021.

[16] L. Josipović, A. Bhattacharyya, A. Guerrieri, and P. Ienne. Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 197–205, Tianjin, Dec. 2019.

[17] L. Josipović, P. Brisk, and P. Ienne. An out-of-order load-store queue for spatial computing. *ACM Transactions on Embedded Computing Systems*, 16(5s):125:1–125:19, Sept. 2017.

[18] L. Josipović, R. Ghosal, and P. Ienne. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 127–36, Monterey, CA, Feb. 2018.

[19] L. Josipović, A. Guerrieri, and P. Ienne. Dynamatic: From C/C++ to Dynamically Scheduled Circuits. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 1–10, Seaside, CA, Feb. 2020.

[20] L. Josipović, A. Guerrieri, and P. Ienne. From C/C++ Code to High-Performance Dataflow Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(7):2142–55, July 2022.

[21] L. Josipović, A. Marmet, A. Guerrieri, and P. Ienne. Resource sharing in dataflow circuits. In *Proceedings of the 30th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 1–9, New York, May 2022.

[22] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 186–96, Seaside, CA, Feb. 2020.

[23] J. Lamoureux and S. J. Wilton. Activity Estimation for Field-Programmable Gate Arrays. In *Proceedings of the 16th International Conference on Field-Programmable Logic and Applications*, pages 1–8, Madrid, Spain, Aug. 2006.

[24] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proceedings of the 19th International Symposium on Code Generation and Optimization*, pages 2–14, Virtual, Feb. 2021.

[25] D. Lee, L. K. John, and A. Gerstlauer. Dynamic Power and Performance Back-Annotation for Fast and Accurate Functional Hardware Simulation. In *Proceedings of the 18th Design, Automation and Test in Europe Conference and Exhibition*, pages 1126–1131, Grenoble, France, Mar. 2015.

[26] R. Li, L. Berkley, Y. Yang, and R. Manohar. Fluid: An Asynchronous High-level Synthesis Tool for Complex Program Structures. In *Proceedings of the 27th International Symposium on Asynchronous Circuits and Systems*, pages 1–8, Beijing, Sept. 2021.

[27] Z. Lin, Z. Yuan, J. Zhao, W. Zhang, H. Wang, and Y. Tian. PowerGear: Early-Stage Power Estimation in FPGA HLS via Heterogeneous Edge-Centric GNNs. In *Proceedings of the 25th Design, Automation and Test in Europe Conference and Exhibition*, pages 1341–6, Virtual, Mar. 2022.

[28] Z. Lin, J. Zhao, S. Sinha, and W. Zhang. HL-Pow: A Learning-Based Power Modeling Framework for High-Level Synthesis. In *Proceedings of the 25th Asia and South Pacific Design Automation Conference*, pages 574–580, Beijing, China, Jan. 2020.

[29] J. Liu, C. Rizzi, and L. Josipović. Load-Store Queue Sizing for Efficient Dataflow Circuits. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 1–9, Hong Kong, Dec. 2022.

[30] A. Mahapatra, Y. Liu, and B. C. Schafer. Accelerating cycle-accurate system-level simulations through behavioral templates. *Integration*, 62:282–291, June 2018.

[31] M. Makni, S. Niar, M. Baklouti, and M. Abid. HAPE: A high-level area-power estimation framework for FPGA-based accelerators. *Microprocessors and Microsystems*, 63:11–27, 2018.

[32] R. Marculescu, D. Marculescu, and M. Pedram. Switching Activity Analysis Considering Spatiotemporal Correlations. In *Proceedings of the 13th International Conference on Computer-Aided Design*, pages 294–99, San Jose, CA, Nov. 1994.

[33] F. Najm. Transition Density: A New Measure of Activity in Digital Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(2):310–323, 1993.

[34] L.-N. Pouchet. *Polybench: The polyhedral benchmark suite*, 2012.

[35] C. Rizzi, A. Guerrieri, P. Ienne, and L. Josipović. A comprehensive timing model for accurate frequency tuning in dataflow circuits. In *Proceedings of the 22nd International Conference on Field-Programmable Logic and Applications*, pages 375–83, Belfast, UK, Aug. 2022.

[36] R. Sarkar and C. Hao. LightningSim: Fast and accurate trace-based simulation for High-Level Synthesis. In *Proceedings of the 31st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 1–11, Marina Del Rey, CA, USA, July 2023.

[37] R. Townsend, M. A. Kim, and S. A. Edwards. From functional programs to pipelined dataflow circuits. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 76–86, Austin, TX, Feb. 2017.

[38] J. Xu, E. Murphy, J. Cortadella, and L. Josipović. Eliminating Excessive Dynamism of Dataflow Circuits Using Model Checking. In *Proceedings of the 31st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 27–37, Monterey, CA, Feb. 2023.

[39] Z. Zhang, D. Chen, S. Dai, and K. Campbell. High-level Synthesis for Low-Power Design. *IPSJ Transactions on System LSI Design Methodology*, 8:12–25, Feb. 2015.

[40] W. Zuo, W. Kemmerer, J. B. Lim, L.-N. Pouchet, A. Ayupov, T. Kim, K. Han, and D. Chen. A Polyhedral-based SystemC Modeling and Generation Framework for Effective Low-power Design Space Exploration. In *Proceedings of the 34th International Conference on Computer-Aided Design*, pages 357–364, Austin, TX, USA, Nov. 2015.