Unleashing Parallelism in Elastic Circuits with Faster Token Delivery

Ayatallah Elakhras*, Andrea Guerrieri*, Lana Josipović† and Paolo Ienne*
*Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland
†ETH Zurich, Department of Information Technology and Electrical Engineering, Zurich, Switzerland

***Abstract*—High-level synthesis (HLS) is the process of automatically generating circuits out of high-level language descriptions. Previous research has shown that dynamically scheduled HLS through elastic circuit generation is successful at exploiting parallelism in some important use-cases. Nevertheless, the literal conversion of a standard compiler's control-data flow graph into elastic circuits often produces circuits with notable resource demands and inferior performance. In this work, we present a methodology for generating more area- and timing-efficient elastic circuits. We show that our strategy results in significant area and timing improvements compared to previous circuit generation strategies.**

I. Introduction

In the last few years, there has been some interest in *high-level synthesis* (HLS) tools converting imperative languages (typically C or C++) into dynamically scheduled circuits [1], [2]. Contrary to classic commercial tools, whose circuits execute carefully precomputed schedules, these tools produce elastic (i.e., dataflow) circuits—that is, perfectly synchronous circuits where every piece of data is accompanied by handshaking signals. Components then execute (i.e., *fire*) operations when and only when all operators are available, without any predefined schedule. Proponents of dynamically scheduled HLS argue that this method is advantageous over its statically scheduled counterpart due to its runtime mechanism adapting to the irregularity and unpredictability of the control structure of a program. This results in performance gain with less code refactoring, for wider classes of applications, and at a generally modest (or at least acceptable) hardware overhead [3].

Elastic circuits bear a strong resemblance to the programming paradigm of the dataflow machines developed in the eighties and early nineties, such as the Sandia National Lab ϵpsilon-2 [4], the MIT Tagged-Token Dataflow [5], or the MIT Monsoon [6]. Although back then the problem was to produce software code for these machines, while here the issue is to generate efficient circuits aggressively exploiting parallelism, some similarities are striking. In particular, while dataflow researchers were often interested in concurrent programming languages, some authors did explore the compilation for dataflow machines of imperative languages (usually FORTRAN) [7], [8]. The goal of this paper is to show that most of the analysis done on producing code for dataflow machines can serve of inspiration to produce significantly better elastic circuits than those generated thus far.

II. The Token Delivery Problem

In this section, we describe the behavior and functionality of elastic circuits. We then illustrate the limitations of existing elastic circuit generation techniques and motivate our work.

A. Elastic Circuits

The basic characteristic of elastic circuits is that tokens flow from one component (*producer*) to another component (*consumer*) through an elastic *channel*, as quickly as data availability and readiness of the consumer allows. Such a flow is insensitive of the actual latency of the producer and consumer components (e.g., of the response time of a memory access). A *token* abstracts a data signal accompanied by handshake control signals that represent the validity of data and the readiness of the receiver, as shown in Figure 1. The main intuition which makes these circuits viable is that tokens do not need to be *tagged* as long as they remain in order within each channel: the order is sufficient to identify which piece of data each token represents (e.g., to which iteration of a loop it belongs). Figure 1 illustrates the functionality of various elastic components. Note an important particularity of MUX which will be relevant in the sequel: While all other components, when triggered, consume a token on *every* input, MUX always consumes a token on the control input and one on the selected input but leaves a token waiting, if any is available, on the input not selected.

B. From Imperative Code to Elastic Circuits

The starting point of elastic circuit generation out of an imperative language such as C/C++ is the intermediate program representation of a compiler, captured in *data flow graphs* (DFGs) and a *control flow graph* (CFG). Nodes in the CFG are blocks of code, referred to as *basic blocks* (BBs); they are connected by control flow edges representing the control decisions. A DFG of a BB is a directed graph of operations connected by edges which indicate data dependences. Dynamically-scheduled HLS techniques [1], [2] directly translate the DFGs of individual BBs into separate circuits and then connect the live-out channels of each BB (i.e., channels carrying tokens that may be used in a later BB) with the live-in channels of the adjacent BBs (i.e., channels potentially requiring tokens generated in earlier BBs).

Figure 2b shows a portion of the elastic circuit implementing the code of Figure 2a and built with conventional techniques [2]; the shown circuit propagates the value of variable x from its producer (assignment operation in BB0) to its consumers (store operation in BB2 and addition in BB5). The token of x exits from each BB through a BRANCH and enters each BB through a MUX. Whenever a token has multiple consumers within a BB, it is replicated using a FORK. It is important to note that the circuit in the figure contains a network of dataless components (shown in orange).

1946-1488/22/$31.00 ©2022 IEEE
DOI 10.1109/FPL57034.2022.00046

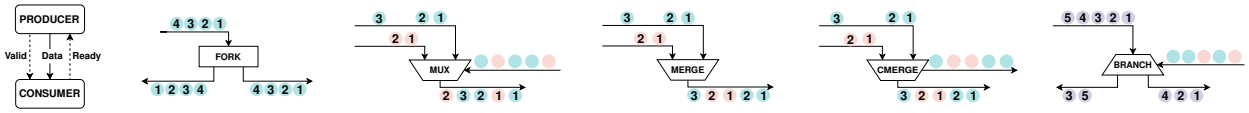


Fig. 1: Elastic Protocol and Classic Elastic Components [3]. Note that the output of MERGE depends also on the arrival times of the inputs. On the other hand, the behaviour of MUX, in particular, and of all other components is completely determined by the order and, where appropriate, the values of the tokens.

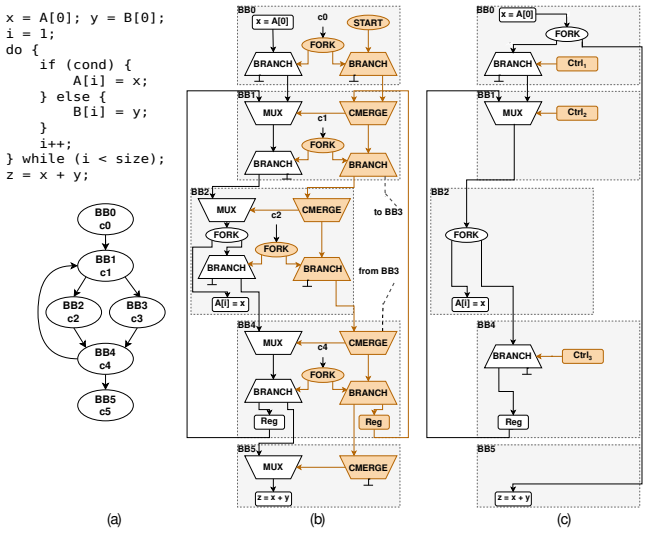


Fig. 2: An Example of Elastic Circuits. (a) A simple code with its CFG. (b) A slightly simplified circuit as produced by state-of-the-art techniques, strictly coupling data propagation with control flow [3]. (c) A more efficient circuit which only needs three control signals.

This network carries a single token that propagates through the circuit strictly following the control flow of the program. The CMERGEs of this network receive a token at only one of their inputs at a time; they send the information of the input origin to all MUXes of the same BB, thus indicating to them which input token they must consume next.

In addition to the components shown in the figure, elastic circuits require buffers (i.e., registers) to achieve correct operation (by removing combinational cycles in loops) and to maximize throughput (by placing buffers to prevent token stalls). As their role and positioning in the circuit have been comprehensively addressed by Josipović et al. [9], we omit them from the rest of this discussion.

C. Fast Token Delivery

The inspection of Figure 2b immediately suggests that this circuit is unnecessarily complicated and most components, albeit simple, are redundant. The redundancy is a result of the conservatism of the control circuit that forces BBs to trigger one at a time and in order, thus limiting the parallelism between different operations, as others have noted before us [10]. Figure 2c shows a simpler and functionally equivalent circuit. Firstly, one can notice that the value of `x` produced at the beginning of the program can be directly delivered to the adder in BB5 without passing through BB1, BB2, BB3, and BB4. Yet, the same direct delivery of `x` is not possible from BB0 to BB2, because it is not certain that the store will ever be executed: In elastic circuits, every token delivered must be consumed and no token must be delivered unless the operation will eventually consume it; therefore, a direct delivery is erroneous. This is why `x` needs to go through the branch so that the corresponding token is suppressed in case `cond` is *false*. Secondly and similarly, although `x` could be delivered to the store node almost directly via a single branch (which simplifies dramatically the implementation by removing many unnecessary components), there is another difficulty: Only a single `x` token is produced in the initial BB whereas the store might be executed many times. Again, in elastic circuits there must be a perfect match between the number of tokens produced and the tokens consumed. The role of the leftmost loop around BB2 in Figure 2c is to *regenerate* the `x` token exactly as many times as BB2 is executed.

The improved circuit of Figure 2c does not only save unnecessary components but also exposes significantly more parallelism that the circuit can exploit. For instance, BB5 can now be executed almost immediately after the circuit starts execution and well before the loop exits.

The contribution of this paper is to show how to obtain systematically circuits like the one shown in Figure 2c by delivering every token as directly as possible from the producers to the respective consumers. We will also show how to compute the control signals in orange without the need of the in-order control network shown in Figure 2b that is largely responsible of limiting the available parallelism.

III. Background on Program Analysis

As suggested earlier, our work is related to early efforts of generating optimized code for experimental dataflow machines. A significant piece of work on that front is the program representations developed by Ballance et al. [7] and refined by Campbell et al. [11]. Our work is heavily inspired by their work and we use some of the representations they introduced. Yet, our purpose is to generate circuits and not code, so in some respects our goals diverge and we use constructs better amenable to efficient hardware implementation.

A. Control Graph Representations

In a CFG, each BB has at most two successors (*true* successor and *false* successor) [12]. The CFG captures the global order of the program, which forces the sequencing of the execution of operations within different BBs.

The *control dependence graph* (CDG) [13] is an extension of the standard CFG. It captures the necessary sequencing between different BBs by identifying the BBs taking control decisions that might prevent the control flow from arriving to some other BBs; this relation is called *control dependence*.

Such relations are based on the theory of postdominator analysis, where a BB_i is postdominated by a BB_j if every path from BB_i to the exit of the graph passes through BB_j [12]. A CDG is constructed by the recursive application of this idea. Figure 4b shows a CFG and Figure 4c shows its corresponding CDG. Loops in the CDG are the minimum strongly-connected subgraphs [13]. In most compilers, a loop is identified by its backward edge. The *loop header* is the target of the *backward edge*, the *loop body* are all BBs in the strongly-connected subgraph, and *loop exits* are any BBs having an edge emerging from the strongly-connected subgraph.

B. Single Assignment Representations

Most standard compilers produce intermediate representations following the *static single assignment* (SSA) form by Cytron et al. [14], which provides a representation of the data flow properties of the program in a way that assures that every variable use in the program has a single reaching definition. If a variable has multiple reaching definitions (at confluence points in the control flow), ϕ-functions are inserted to *merge* these definitions into a single definition. SSA-form does not include control information; it provides no distinction between the various definitions reaching a given ϕ-function. Thus, they are naturally translated into a MERGE circuit component, which is a nondeterministic component [2].

Ballance et al. [7] introduced the *gated single assignment* (GSA) form that resolves the limitation of the SSA. It extends the semantics of the ϕ-function to capture and hold control predicates that determine which definition is relevant depending on control flow decisions. GSA adds three types of gating functions, denoted by γ, μ, and η, to account for various control-flow situations.

IV. Token Delivery without Loops

In this section, we describe our proposed methodology of elastic circuit generation for faster token delivery. We start analyzing control structures excluding loops, by initially skipping backward edges. We include them back in Section V.

A. From Basic Blocks to Subcircuits

Our first step is to translate operations of individual BBs into elastic components and to connect the data-dependent components within each BB. As for conventional techniques, such as the work by Josipović et al. [2], this is a trivial step because, essentially, every instruction in the BB corresponds directly to an elastic component and data-dependent components can be connected directly. The only important difference with the method of Josipović et al. is that we start from the GSA-form and not the SSA-form, hence there are no ϕ nodes (that is, no MERGE components in the obtained circuits). Rather, there are gating functions that can be translated to MUXes, where the predicate of the gated function holds information about the select signal of MUX.

Although information about the select signals of MUXes is readily provided by the gated functions predicates, we need to do some necessary analysis prior to connecting the select signal of MUXes, as we will explain in Section IV-D.

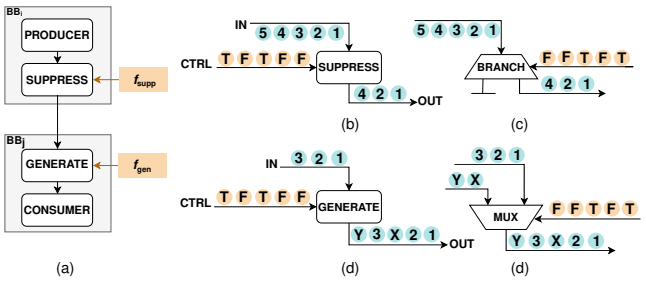


Fig. 3: Connecting Producers to Consumers. (a) The generic way we connect any producer to any consumer; depending on the conditions, SUPPRESS and GENERATE may in fact disappear. (b–c) The functionality of SUPPRESS with its natural implementation. (d–e) The functionality of GENERATE with its natural implementation.

B. Connecting Subcircuits

We deliver tokens from producers (assignments of a new variable in GSA representation) to consumers (users of the assigned variable). Since we have already completely implemented each BB independently, we are only left with data-dependent components across different BBs. The GSA representation guarantees that there will be only one producer for one or more consumers (i.e., only one definition for each use). Our analysis considers independently each producer-consumer pair to generate control components and their conditions; this will, naturally, generate a number of potentially redundant components (e.g., in case there are multiple consumers that could share the same components). We remove these redundancies with peephole optimizations in a later step, prior to generating the circuit.

The important problem we need to handle is that, in general, at one point in time, there is no guarantee that the control flow will arrive to any of the two BBs containing the producer and consumer operations. This implies that, independently, a token may or may not be issued from the producer and a token may or may not be absorbed by the intended consumer. As discussed before, a fundamental property of our circuits is that every produced token must be consumed and every consumed token must be produced. Hence, our goal is to match the conditions of production (definition) of a token with the conditions of consumption (use) of the token.

We, therefore, connect every producer to the corresponding consumer through two new elastic components, as shown in Figure 3a. The SUPPRESS component has the functionality shown in Figure 3b and simply eliminates a token from the sequence when the Boolean control signal (CTRL) receives a token *true*. The GENERATE component is shown in Figure 3d and simply inserts a token whose value is undetermined (arbitrarily a '0' or '1') every time the Boolean control signal (CTRL) receives a token *true*; every time the control signal gets a token *false*, a token from the input stream is propagated to the output. These two new components are readily generated with ordinary elastic components, as Figure 3c and e illustrate. As Figure 3a suggests, the SUPPRESS component fires every time the BB of the producer executes, thus essentially belongs to it; conversely, the GENERATE component must fire every time the BB of the consumer executes. Clearly, appropriately

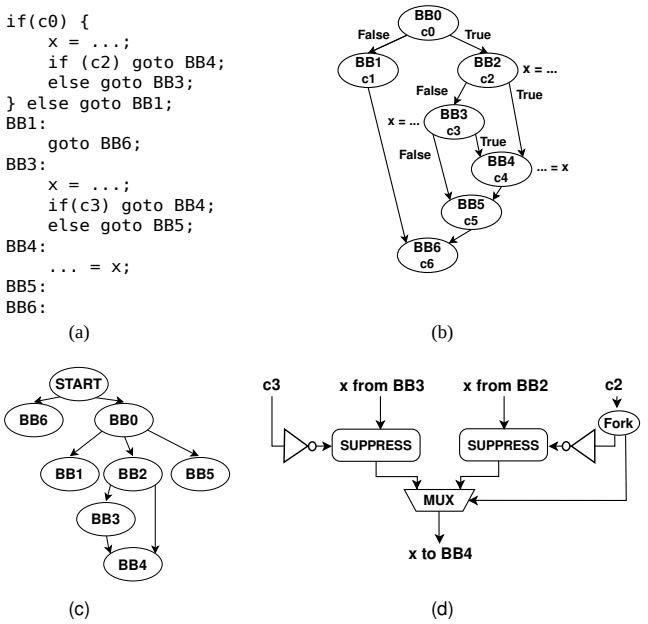


Fig. 4: Delivery Problem. (a) Sample code skeleton that has a complicated control structure generated by `goto` statements or similar constructs. (b) CFG for the given code with two producers and one consumer of variable x. (c) CDG for the given CFG. START is the entry node in the graph. (d) Optimized delivery circuit.

controlling the SUPPRESS and GENERATE components is sufficient to ensure that the tokens emitted by the producer match the expectations of the consumer; we address this in the next section. It is worth mentioning that, in a correct program, a GENERATE component should not be required between a producer and consumer in the datapath; having a consumer absorbing a token that the producer has not generated is nonsensical. However, it could be, potentially, required in the control path, as we will explain in Section IV-D. We consider the general case and compute the control signal of GENERATE for any pair of producer and consumer, which happens to deliver only *false* control signals for producers and consumers in the datapath; we rely on logic synthesis to remove away the logic of GENERATE in this case.

Up to this point, our analysis matches the analysis done by Ballance et al. [7]. However, their analysis could insert, potentially, several SUPPRESS nodes (they call them *switch* nodes) between a producer-consumer pair, where each SUPPRESS node takes a simple control signal. Our analysis, on the contrary, introduces exactly a single SUPPRESS node in the data path which takes a more complex condition, as we will explain in the coming two sections. The example of Figure 5 discussed in Section IV-D will clarify the difference between these approaches and the advantages of ours.

C. Generating and Suppressing Tokens

The goal of this section and the next is to implement the delivery path between an arbitrary pair of a producer and consumer operations (Figure 3a). In this section, we compute the logic conditions to suppress or generate a token; in the next section, we solve the problem of computing and delivering the tokens corresponding to such conditions. To illustrate each step of the algorithm, we apply our algorithm to the delivery problem in the CFG in Figure 4b that requires the delivery of the variable x from its two producers in BB2 and BB3, respectively, to its consumer in BB4.

ALGORITHM 1 (Computation of Delivery Conditions). For each pair of producer-consumer pair, we compute as follows the logic conditions f_{supp} and f_{gen}, identifying the situations when a token from the producer needs to be suppressed and when a token for the consumer needs to be generated, respectively.

1) **Identify Control Dependencies of Producer and Consumer BBs.** Using the CDG, compute two sets $\mathcal{S}_{\text{prod}}$ and $\mathcal{S}_{\text{cons}}$, representing the set of BBs that the producer BB and the consumer BB, respectively, are control dependent on. For our example, whose CDG is shown in Figure 4c:
   a) (Producer in BB2, Consumer in BB4) $\mathcal{S}_{\text{prod}} = \{\text{BB0}\}$ and $\mathcal{S}_{\text{cons}} = \{\text{BB0}, \text{BB2}, \text{BB3}\}$.
   b) (Producer in BB3, Consumer in BB4) $\mathcal{S}_{\text{prod}} = \{\text{BB0}, \text{BB2}\}$ and $\mathcal{S}_{\text{cons}} = \{\text{BB0}, \text{BB2}, \text{BB3}\}$.
2) **Eliminate Common Control Ancestors.** Adjust the sets $\mathcal{S}_{\text{prod}}$ and $\mathcal{S}_{\text{cons}}$ by removing the common elements $\mathcal{S}_{\text{prod}} \cap \mathcal{S}_{\text{cons}}$ from each set. This is equivalent to removing the common ancestor BBs between the producer and consumer BBs in the CDG, such as BB0 in our example. The decision of such common BBs plays no role in generating or suppressing tokens because either both the producer and consumer have a chance to execute (determined by other conditions) or none of them can execute. Therefore, they should be dropped from our analysis. In our example:
   a) (Producer in BB2, Consumer in BB4) $\mathcal{S}_{\text{prod}} = \emptyset$ and $\mathcal{S}_{\text{cons}} = \{\text{BB2}, \text{BB3}\}$.
   b) (Producer in BB3, Consumer in BB4) $\mathcal{S}_{\text{prod}} = \emptyset$ and $\mathcal{S}_{\text{cons}} = \{\text{BB3}\}$.
3) **Compute Conditions of Production and Consumption.** Compute the Boolean expressions (f_{prod} and f_{cons}) representing the conditions under which the producer will produce a token (if f_{prod} is true) and the consumer will consume a token (if f_{cons} is true). They are calculated by traversing the CDG starting from each BB in the $\mathcal{S}_{\text{prod}}$ to the producer BB, and each BB in $\mathcal{S}_{\text{cons}}$ to the consumer BB. Each path identifies a Boolean product of elementary conditions expressing the reaching of the target BB from the corresponding member of the set; the products for all such paths are added. Empty control dependency sets result in a true Boolean expression. In our example:
   a) (Producer in BB2, Consumer in BB4) $f_{\text{prod}} = 1$ and $f_{\text{cons}} = c2 + \overline{c2} \cdot c3$.
   b) (Producer in BB3, Consumer in BB4) $f_{\text{prod}} = 1$ and $f_{\text{cons}} = c3$.
4) **Adjust the Consumption Condition for MUXes.** Correct the expression f_{cons} if the consumer is a MUX by multiplying f_{cons} by either the value of the MUX

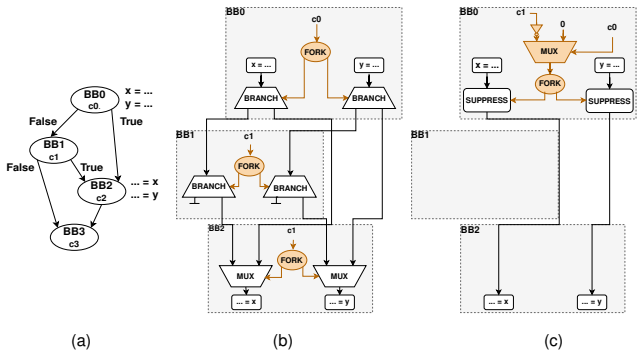


Fig. 5: Delivery Problem and Two Valid Solutions. (a) CFG with two pairs of producers and consumers in identical BBs. (b) A valid solution that does not minimize the data path by using one SUPPRESS per basic block instead of one per producer-consumer channel. (c) Our proposed solution that minimizes the data path at the cost of a slightly more complicated control path; yet, the control path can be shared among different data paths.

select signal (f_{sel}) or its complement, as appropriate, to force the consumption condition to be true *only* if f_{sel} activates the MUX input connected to the producer currently under study. This adjustment should occur only if there is a possibility that the consumer executes when the input connected to its producer is not activated, i.e., if $f_{\text{cons}} \cdot \overline{f_{\text{sel}}}$ evaluates to nonzero. Also, this adjustment has to be performed only if the producer BB is not control dependent on the BB whose condition is the select signal of the MUX. In our example, $f_{\text{sel}} = c2$ for the producer in BB2 and $f_{\text{sel}} = \overline{c2}$ for the producer in BB3.

a) (Producer in BB2, Consumer in BB4) $f_{\text{prod}} = 1$ and $f_{\text{cons}} \cdot \overline{f_{\text{sel}}} = (c2 + \overline{c2} \cdot c3) \cdot \overline{c2} = \overline{c2} \cdot c3$ which is nonzero; therefore, adjust the consumption condition to $f_{\text{cons}} = c2 \cdot (c2 + \overline{c2} \cdot c3) = c2$.
b) (Producer in BB3, Consumer in BB4) $f_{\text{prod}} = 1$ and $f_{\text{cons}} = c3$. Since BB3 is control dependent on BB2, f_{cons} is unmodified.

5) **Conditions of Suppression and Generation.** Compute the suppression and generation conditions as $f_{\text{supp}} = f_{\text{prod}} \cdot \overline{f_{\text{cons}}}$ and $f_{\text{gen}} = \overline{f_{\text{prod}}} \cdot f_{\text{cons}}$. In other words, we need to suppress a token if the producer outputs one but the consumer will not absorb any, and we need to generate one if the consumer absorbs it and the producer does not output it. In our example:

a) (Producer in BB2, Consumer in BB4) $f_{\text{supp}} = 1 \cdot \overline{c2} = \overline{c2}$ and $f_{\text{gen}} = \overline{1} \cdot c2 = 0$.
b) (Producer in BB3, Consumer in BB4) $f_{\text{supp}} = 1 \cdot \overline{c3}$ and $f_{\text{gen}} = \overline{1} \cdot c3 = 0$.

The tokens used to control SUPPRESS and GENERATE must have the Boolean value expressed by f_{supp} and f_{gen}, respectively. In simple situations, such as our example, this already results in the correct circuit shown in Figure 4d. Yet, in the general case, there are some issues to take care of to correctly produce these tokens, as shown in the next section.

D. Delivering Control Tokens

We now have to generate control tokens, both for the MUXes resulting from the conversion of the GSA BBs (Section IV-A) and for the SUPPRESS and GENERATE components (Section IV-C). Consider the CFG of Figure 5a, that represents a program composed of nested `if-then-else` statements. The tokens of `x` and `y` produced in BB0 and consumed in BB2 result in $f_{\text{supp}} = \overline{c0} \cdot \overline{c1}$. It would be tempting to simply compute this function with an elastic circuit: in our case, a simple elastic NOR gate between $c0$ and $c1$ ("unless either $c0$ is *true* or $c1$ is *true*, suppress the token produced in BB0"). Logically, when $c0$ is *true*, the output is *false* and the result is apparently correct. Yet, there is a problem summarized in the following table:

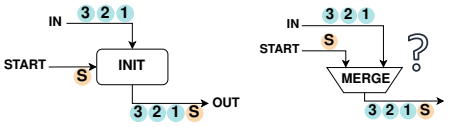
$c0$	$c1$	f_{supp}
false	*false*	*true*
false	*true*	*false*
true	(no token)	*false*

If $c0$ is *true*, $c1$ will never be generated, because BB1 does not execute; consequently, the NOR gate will not receive both tokens in this case and will never compute the result. The condition for the SUPPRESS node will never arrive and the circuit will deadlock.

A simple solution is to implement, as mentioned above, the logic function with elastic logic gates but, instead of connecting the conditions directly, consider them as an ordinary producer and consumer relation and follow recursively the procedure of Section IV-C. This will result in the insertion of GENERATE components to compensate for the conditions that will never arrive by inserting a number of arbitrary '0' or '1' tokens that are perfectly unnecessary except to fire the appropriate components. Although the solution is functionally correct, it generates and consumes pointless tokens. We, therefore, follow a different strategy to generate control tokens, which is functionally equivalent but more economical: it ignores missing tokens through MUXes which do not consume them instead of generating pointless ones.

ALGORITHM 2 (Delivery of Control Tokens). For each logic condition $f(l_0, l_1, ...l_n)$ (e.g., f_{supp}), we determine the corresponding circuit with the following algorithm.

1) **Partial Ordering of the Literals.** Create a graph $\mathcal{G}_{\text{ord}}$ where the nodes are all the literals l_i in f and where there is an edge between two literals if in the CDG there is a path between the BBs of the corresponding literals. Intuitively, this indicates that a control token corresponding to the literal at the head of an edge will be available only under some conditions determined by the value of the literal at the tail (whose corresponding control token may or may not be available based on the incoming edges of that literal). For the example in Figure 5, $\mathcal{G}_{\text{ord}}$ is trivial: there are only two literals $c0$ and $c1$ and there is an edge $c0 \rightarrow c1$, thus $c1$ is available only under some condition about $c0$.
2) **Order the Literals.** Pick repeatedly any literal l from $\mathcal{G}_{\text{ord}}$ among those without an incoming edge, remove it from $\mathcal{G}_{\text{ord}}$, and add it to a list $\mathcal{L}_{\text{ord}}$which will hold an ordered list of the literals. Since the first literal in the list has no predecessors in the whole $\mathcal{G}_{\text{ord}}$, its



(a) (b)

Fig. 6: The INIT Component. (a) Before simply propagating all input tokens to the output, this new component waits for a single token on the start input and propagates this first to the output. (b) A naive and unsafe implementation of the INIT component; if the start token arrives strictly before any other input token (which may be guaranteed in some real circuits), this implementation behaves correctly.

corresponding token will be certainly available. Tokens corresponding to literals appearing later in the list will be available under conditions identified exclusively by literals preceding them. In the example of Figure 5, there is only one possibility to build the list and it results into $\mathcal{L}_{\text{ord}} = \{c0, c1\}$.

3) **Successive Shannon Expansions.** Use the order of the literals in $\mathcal{L}_{\text{ord}}$ to perform successive Shannon expansions of f. Assuming, without loss of generality, that the literals in $f(l_0, l_1, ...l_n)$ are in the order of $\mathcal{L}_{\text{ord}}$ (if they are not, the literals can be renamed), we write

$$f = \mathsf{MUX}(l_0, f_{l_0}(l_1, ...l_n), f_{\overline{l_0}}(l_1, ...l_n)) = ...$$

and we repeat the expansion for all literals. We are simply implementing the Boolean function of f with MUX components and, thanks to the property of MUXes that they do not consume tokens on the deselected inputs, we do not need to worry about missing tokens. In our example, $f_{\text{supp}} = \overline{c0} \cdot \overline{c1} = \mathsf{MUX}(c0, \overline{c1}, 0)$.

Figure 5c shows the circuit produced by our algorithm. It is clearly simpler and more straightforward than the circuit in Figure 5b which is what we would obtain if we use the techniques of Ballance et al. [7] to generate a circuit.

V. Token Delivery with Loops

The introduction of loops through the consideration of the backward edges brings two new problems: (1) On the one hand, there is a new set of MUX nodes when one of the producers of a particular value is carried by the loop (i.e., loop-carried dependencies). Authors have recognized that MUXes cannot follow the same rules as for dependencies outside of loops (Section IV-B) [11]. (2) The token miscount is no longer limited to a single token that may or may not be produced and may or may not be consumed; now we may have many tokens generated during loop execution but only the last one might be consumed and, in a dual way, we may have single tokens which are used many times. An example of the latter case is the fact that x is used in several loop iterations in our example of Figure 2. The next two sections address these two issues.

A. Controlling MUXes in Loops

Consider the MUX created as a result of the backward edge of a loop, such as the one at the input of BB1 in Figure 2c where x gets reassigned in each loop iteration. The MUX should take the value of x arriving from outside of the loop in the first iteration and thereafter should take the local value of x *regenerated* from within the loop, until the loop completes execution. In the general case, we can have some inner loop in a nested loop structure where the MUX should take a fresh value from outside *every* time a new outer loop iteration starts and thereafter should take the local inner value, until the loop completes execution. It is fairly easy to prove that the MUX select signal is the same as the exit condition of the loop but delayed by one cycle. The only particularity is the first loop iteration ever, for which no loop exiting condition has ever been computed; in this case, we need to force the selection of the value of x arriving from outside. In summary, the sequence of MUX select signals is a token that says "from outside" followed by tokens representing the loop condition with appropriate sign (i.e., "continue loop" = "from inside" and "exit loop" = "from outside").

Ballance et al. [7] essentially made the same observation when they introduced the μ-function in GSA instead of the normal γ-function (our MUX) for loops. The behavior of μ is fairly complex and was refined in a later paper [11]. We implement the very same functionality by introducing a component INIT which does exactly what is described above (see Figure 6a): it inserts a token before the stream of conditions coming on the input channel. This component is more tricky to implement and Figure 6b shows a naive implementation. If the token on the start input arrives before any token on the other input, this implementation is correct because the potential nondeterminism of the MERGE behaviour has no impact; this is arguably a typical case, but, as Campbell et al. observed, it is not always true. It is not clear whether a correct implementation of INIT is possible using only classic elastic components, but it is straightforward to implement it directly by writing the corresponding state machine (it is clear that this component is sequential because it needs to "remember" whether the start token has already been sent). We omit the details of the implementation due to the lack of space.

In summary, the circuit we need to control the new MUXes introduced by the back edges (the μ-functions of Ballance et al. [7]) is as follows: (i) we create the logic function expressing whether we are exiting the loop (i.e., if the loop has multiple exits, this will be the logic OR of the various exit conditions); (ii) we implement this logic function with Algorithm 2 (Section IV-D); and (iii) we feed this signal to the select input of the MUX through an INIT component.

B. Producers inside Loops and Consumers outside

Now we need to take care of the miscount of tokens. We start from the situation when the producer is part of a loop and the consumer is not; in this case, only the last token produced before the exit must be delivered. Consider, for instance, the example of Figure 7a and the case of y which is produced in the inner loop (BB3) and consumed outside the nested loop (BB8). The slight complexity is due to the fact that the producer is deep inside a nested loop structure with some loops having several exiting edges and from BBs before or after the BB of the producer. The solution is in fact simple and one can look at this case as a normal case of token delivery *every time the producer is executed*: for this we cut

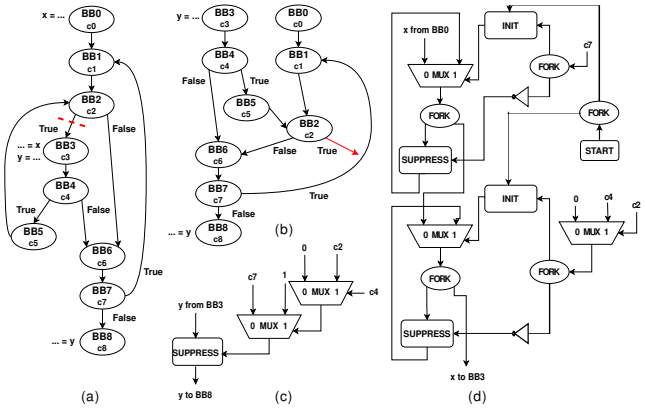


Fig. 7: Producers and Consumers across Nested Loop Borders. (a) A producer outside the loop with the consumer inside (x) and a producer inside with the consumer outside (y). (b) Modified CFG by breaking the cycle before the producer to implement the delivery from inside the loop to the consumer outside it. (c) Circuit for delivering the token in variable y by analyzing the CFG in (b). (d) Circuit for delivering the token in variable x.

the edges entering the BB of the producer, so that the producer is no longer in a loop (Figure 7b) and, thus, the delivery from producer to consumer can be simply implemented as described in Section IV. One can note that this simple procedure avoids considering separately `while` and `do...while` loops: it works for an arbitrary loop structure. The circuit shown in Figure 7c is a result of applying Algorithms 1 and 2 on the modified CFG of Figure 7b.

C. Producers outside Loops and Consumers inside

The dual problem is when the producer is outside a loop and the consumer is inside it; in this case, only one token is sent but many copies need to be consumed. Consider the case of x in Figure 7a: x is produced before entering the loop (BB0) and consumed inside the loop (BB3). Once the problem is detected by elementary loop analysis, the solution is, simply, applying the GSA form algorithm for managing loop-carried dependency by imagining the dependency to be a "reassignment" (x = x in our case) right after the consumer [7], [11]. This creates a second producer, this time inside the loop. Because there are now two definitions to the variable, this would naturally result in MUXes inserted at the loop headers and controlled as described in Section V-A: the token will be regenerated as frequently as needed. SUPPRESS components are then inserted to stop the regeneration when the loops complete execution. Figure 7d shows the circuit for delivering the token in variable x.

VI. Experimental Results

We integrated our proposed HLS methodology into *Dynamatic* [17], an open-source C-to-elastic circuits HLS tool based on LLVM [18]. We replaced key parts of *Dynamatic*'s elastic circuit generation strategy with a custom LLVM pass implementing our methodology. We use the unmodified backend of *Dynamatic* to insert buffers and generate the RTL description of the circuit. We synthesize the generated VHDL netlists with Vivado [19] with a clock period timing constraint of 4 ns, targeting a Kintex-7 Xilinx FPGA. We simulate the designs with ModelSim [20] and use a set of test vectors for functional verification. We measure (1) the cycle count obtained from simulation, (2) the clock period (CP) from the postrouting timing analysis, and (3) resource usage (i.e., LUT, FF and DSP counts) reported from Vivado after placement and routing.

TABLE I: Description of the Benchmarks.

Benchmark	Description
Stencil2d	2-dimensional Stencil computation
PolyRqmul	Polynomial multiplication of post-quantum cryptography [15]
BinSearch	Divide and conquer array search algorithm
Sobel	Ordinary Sobel filter
Gaussian	Ordinary Gaussian filter
Matvec	Matrix-vector multiplication
Bicg	Subkernel of the *BiCGStab* linear solver [16]
BinGCD	Stein's binary algorithm for greatest common divisor (GCD)
Fir	Ordinary finite impulse response (FIR) filter

Our benchmarks are nine kernels with different control structures, listed in Table I. The memory access patterns of our benchmarks can be resolved at compile-time; thus, these circuits do not require any load-store queues.

Table II summarizes the timing and resources of the circuits generated by our approach (Ours) in comparison to those generated by the baseline *Dynamatic* ([17]). Results show a significant reduction in area and an improvement in performance of our generated circuits. Figure 8 graphically represents our results normalized to those by *Dynamatic*. Advantages are generally quite tangible. On two fronts (resources and critical path), the improvement is largely due to the avoidance of a multitude of elastic components inserted by *Dynamatic* to implement its simpler delivery strategy (we suggested this effect qualitatively in our motivating example of Figure 2). Additionally, our results sport a gain in number of cycles which is often nonnegligible; this is mainly because our analysis and the resulting delivery of tokens exposes more parallelism for the circuits to exploit. We describe here some of these important situations.

Stencil2d, **Sobel**, and **Gaussian** are composed of deep loop nests. Our approach overlaps the outer loops iterations with the execution of the inner loops; thus, exploiting more parallelism. This is in contrast to *Dynamatic* which advances outer loop iterations only when the inner loops complete execution.

PolyRqmul and **BinSearch** are composed of multiple natural loops that are loading and processing different parts of arrays: our approach parallelizes the execution of these loops, while *Dynamatic* sequentializes them. Similarly, **BinGCD** involves iterative bit-level analysis on two operators: our approach allows for parallelizing the analysis on the two operators. **BinGCD** runs here with 32-bit operators and thus the execution takes only a small number of cycles.

Bicg consists of a loop nest with the outer loop loading one array element that is used by all iterations of the inner loop. Although we can now decouple the memory loads in the outer loop from the processing in the inner loop, data dependencies limit the amount of exploitable parallelism. Yet, our design reduces the resources and is Pareto-dominant to *Dynamatic*.

TABLE II: Elastic circuits produced by our methodology, contrasted to those produced by the open-source tool *Dynamatic* [17]. We measure cycle counts in simulation and obtain the timing and resources from Vivado, after place-and-route.

Benchmark	**Cycle count**		**CP (ns)**		**Execution time (μs)**			**LUTs**			**FFs**			**DSPs**
	[17]	Ours	[17]	Ours	[17]	Ours		[17]	Ours		[17]	Ours		
Stencil2d	12610	7072	7.21	6.78	90.9	48.0	**-47%**	2826	2438	**-14%**	2253	1806	**-20%**	12
PolyRqmul	59221	44696	6.00	5.8	355.3	259.2	**-27%**	2002	1823	**-9%**	1767	1470	**-17%**	6
BinSearch	210	106	6.00	6.40	1.26	0.68	**-46%**	1700	1349	**-21%**	1750	1053	**-40%**	0
Sobel	5556	4279	6.90	6.70	38.3	28.7	**-25%**	4310	3740	**-13%**	4058	2640	**-35%**	12
Gaussian	5858	5838	6.50	6.00	38.1	35.0	**-8%**	2622	2150	**-18%**	2095	1473	**-30%**	3
Matvec	10109	10010	6.80	6.20	68.7	62.1	**-10%**	1240	1115	**-10%**	1061	829	**-22%**	3
Bicg	943	910	6.10	5.70	5.75	5.19	**-10%**	1797	1668	**-7%**	1484	1255	**-15%**	6
BinGCD	13	9	4.58	5.00	0.060	0.044	**-27%**	2381	1480	**-38%**	2220	1063	**-52%**	0
Fir	1010	1008	5.93	4.60	5.99	4.64	**-23%**	672	600	**-11%**	621	515	**-17%**	3

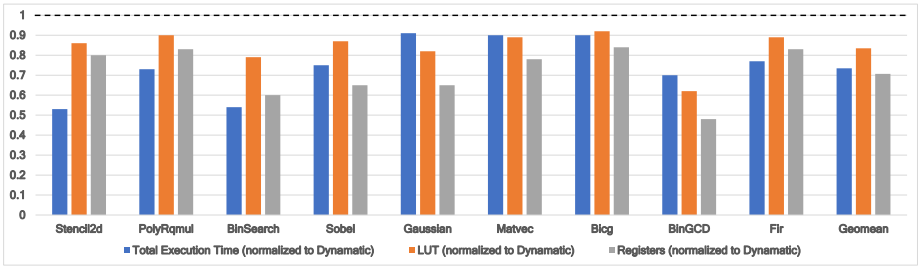


Fig. 8: Results. Execution time and resources of our circuits, normalized to the results of *Dynamatic* [17].

VII. Related Work

We have addressed throughout the manuscript our similarities and differences from classic work aimed at producing code for dataflow machines from imperative code [7], [11]. We focus here on more recent work aimed at producing circuits.

Latency insensitive protocols have been extensively used to construct synchronous and asynchronous elastic circuits [2], [21]–[23]. Several efforts explored the automatic generation of asynchronous circuits from high-level descriptions [24]–[26] and one addressed synchronous circuits [2]. The closest efforts to ours are the works by Budiu et al. [25] and by Josipović et al. [2] which employ the circuit generation strategies of *Dynamatic* [17]. We use it as a baseline but, as our discussion illustrates and our results show, the existing strategy is qualitatively and quantitatively inferior to ours. The works by Zaidi et al. [10] and Li et al. [27] aim at better exploiting instruction level parallelism in dataflow circuits by performing control flow graph analysis and optimizations. Li et al. [27] impose a "canonical form" requirement on the CFGs, which is violated by `goto` statements and, probably, similar constructs such as `continue` and `break` statements. We, on the other hand, do not impose any constraint on the control structures that we support, and our methodology results in simpler datapaths; for the example in Figure 5, the techniques by Li et al. would generate a circuit similar to Figure 5b. Zaidi et al. [10] present Value State Flow Graph, a compiler intermediate representation that represents the control flow in terms of Boolean predicates. However, the work does not detail the calculation and usage of the Boolean predicates in complicated control structures where the predicates should be functions of conditions from multiple BBs such as the examples we discuss in Figure 4 and Figure 5.

VIII. Conclusions

In this paper, we have completely rewritten how to deliver tokens in an elastic circuit obtained from imperative code. We have done so inspired by the program representations developed in the nineties when some authors were trying to efficiently map imperative languages on dataflow machines. In general, we have appreciable gains on three fronts: (i) we save resources by having significantly less components on the paths from producers to consumers; (ii) this naturally also improves combinational delays; and (iii) our faster delivery exposes potentially much more parallelism and thus reduces the number of execution cycles. The latter benefit is, qualitatively, the most important because the truly critical task of an HLS tool is to expose as much parallelism as possible for the resulting circuit to exploit. Yet, most HLS approaches, both statically and dynamically scheduled, are in fact quite limited: Statically scheduled techniques, because of their very static scheduling, cannot really overlap, for instance, inner and outer loop iterations of nested loops or independent sequential program loops—at least, not in the most general case or not without tangible code rewriting. On the other hand, supporting this seems naturally feasible with dynamically scheduled methods; yet, to our best knowledge, circuit generation techniques in this class similarly fail because of the conservative (one may even say *crude*) way they deliver tokens. We hope that this work opens a new dimension to the parallelization of dynamically scheduled circuits automatically extracted from imperative languages like C/C++.

Acknowledgments

This research is partially supported by Huawei.

REFERENCES

[1] 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*, Austin, Tex., Mar. 2005, pp. 177–86.

[2] 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*, Monterey, Calif., Feb. 2018, pp. 127–36.

[3] L. Josipović, A. Guerrieri, and P. Ienne, "Synthesizing general-purpose code into dynamically scheduled circuits," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 97–118, Second quarter 2021.

[4] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes, "The Epsilon dataflow processor," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Apr. 1989, pp. 36–45.

[5] Arvind and R. S. Nikhil, "Executing a program on the MIT Tagged-Token dataflow architecture," *IEEE Trans. Computers*, vol. 39, pp. 300–318, 1990.

[6] G. M. Papadopoulos, "Implementation of a general-purpose dataflow multiprocessor," Ph.D. dissertation, Massachusetts Institute of Technology, Laboratory for Computer Science, 1998.

[7] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstien, "The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages," in *Proceedings of the 11th ACM SIGPLAN Conference on Programming Language Design and Implementation*, White Plains, NY, Jun. 1990, pp. 257–71.

[8] K. J. Ottenstein and S. J. Ellcey, "Experience compiling Fortran to program dependence graphs," *Software: Practice and Experience*, vol. 22, 1992.

[9] 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*, Seaside, Calif., Feb. 2020, pp. 186–96.

[10] A. M. Zaidi and D. Greaves, "A new dataflow compiler IR for accelerating control-intensive code in spatial hardware," in *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, Phoenix, AZ, May 2014, pp. 122–131.

[11] P. L. Campbell, K. Krishna, and R. A. Ballance, "Refining and defining the program dependence web," University of New Mexico, Tech. Rep. Technical Report 93-6, Mar. 1993.

[12] L. Torczon and K. Cooper, *Engineering a Compiler*, 2nd ed. Morgan Kaufmann, 2011.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, pp. 319–349, 1987.

[14] R. G. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of the 16th Symposium on the Principles of Programming Languages*, Austin, TX, Jan. 1989, pp. 25–35.

[15] *NTRU, Algorithm specification and supporting documentation*, NTRU, 2020. [Online]. Available: https://ntru.org

[16] L.-N. Pouchet, *Polybench: The polyhedral benchmark suite*, 2012. [Online]. Available: http://www.cs.ucla.edu/pouchet/software/polybench

[17] 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*, Seaside, Calif., Feb. 2020, pp. 1–10.

[18] *http://www.llvm.org*, The LLVM Compiler Infrastructure, 2018. [Online]. Available: http://www.llvm.org

[19] *Vivado Design Suite*, Xilinx Inc., 2017. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado.html

[20] Mentor Graphics, "ModelSim," 2016. [Online]. Available: https://www.mentor.com/products/fv/modelsim/

[21] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–76, Sep. 2001.

[22] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *Proceedings of the 43rd Design Automation Conference*, San Francisco, Calif., Jul. 2006, pp. 657–62.

[23] 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*, Vienna, Sep. 2017, pp. 175–84. [Online]. Available: http://doi.acm.org/10.1145/3127041.3127055

[24] D. Edwards and A. Bardsley, "Balsa: An asynchronous hardware synthesis language," *The Computer Journal*, vol. 45, no. 1, pp. 12–18, Jan. 2002.

[25] M. Budiu and S. C. Goldstein, "Pegasus: An efficient intermediate representation," Carnegie Mellon University, Tech. Rep. CMU-CS-02-107, May 2002.

[26] S. F. Nielsen, J. Sparsø, J. B. Jensen, and J. S. R. Nielsen, "A behavioral synthesis frontend to the Haste/TiDE design flow," in *Proceedings of the 15th International Symposium on Asynchronous Circuits and Systems*, Chapel Hill, N.C., May 2009, pp. 185–94.

[27] 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*, Beijing, Sep. 2021, pp. 1–8.