Translating General Slack Elastic Programs into Dataflow Circuits

Xiayuan Wen
Yale University
New Haven, CT, USA
xiayuan.wen@yale.edu

Rui Li
Intel
Santa Clara, CA, USA
rui.lee92@gmail.com

Rajit Manohar
Yale University
New Haven, CT, USA
rajit.manohar@yale.edu

***Abstract*—Static dataflow circuits are a common target for asynchronous circuit synthesis. Dataflow circuits can be implemented using fine-grained asynchronous pipelines from a variety of circuit families, typically leading to high-throughput circuit implementations. This approach has been used to design a number of different asynchronous chips including microprocessors, FPGAs, on-chip networks, and arithmetic circuits.**

Existing methods for translating slack elastic programs into dataflow circuits restrict the way channels can be used in an individual process; in particular, they require that a channel can only be used once per process loop iteration. We present a systematic solution to this problem that lifts this restriction, thereby enabling the translation of general slack elastic processes into dataflow components. We compare our approach with alternative strategies for this problem, and discuss why our method is better suited for high-throughput asynchronous dataflow circuit synthesis.

***Index Terms*—asynchronous circuits; dataflow synthesis; shared channels**

I. Introduction

The static dataflow model of computation is a well-established method for designing parallel computation structures. In this model, the computation is described as the parallel composition of a large number of concurrent message-passing dataflow elements that together implement the computation of interest. One of the appealing aspects of dataflow circuits is that it is easy to design *slack elastic* systems—systems where pipelining details can be postponed to a late stage in the design process because one can freely introduce pipelining without changing the calculations performed by the computation [1].

Asynchronous circuits provide a natural realization of static dataflow computing. Dataflow building blocks can be implemented using a wide range of asynchronous circuit families, including QDI circuits [2], micropipelines [3], GasP circuits [4], or MOUSETRAP circuits [5].

There have been many efforts to automate the synthesis of asynchronous circuits from behavioral specifications. One approach that takes a strict approach to circuit generation is syntax-directed translation [6], [7]; this approach respects the sequencing of all operations specified in the behavioral description. Another approach is the dataflow-style decomposition method, which includes data-driven decomposition (DDD) [8] and static token form (STF) [9] that generates dataflow circuits from a message-passing programming language. While the details differ, both DDD and STF rely on the projection theorem [10] to ensure correctness; the projection theorem, in turn, relies on slack elasticity. However, neither DDD nor STF can generate dataflow graphs from a program that includes multiple channel actions at different points in the dynamic execution of a single iteration of the program [8], [9].

As an illustrative example, consider the following simple process described in CHP[1]:

$$*[\ L?x; R!x; L?x; R!x\ ]$$

Technically this process violates the requirements for dataflow generation for both DDD and STF, because there are two L actions and two R actions per loop iteration. We say that both L and R have *repeated channel actions* in this process. To address this issue, the process can be rewritten as an equivalent one with only a single channel action per iteration of the program. For the example shown above, the resolution is trivial since the process is obviously equivalent to $*[L?x; R!x]$, which is simply a one-place dataflow buffer.

As a more complex example, consider the process shown below, in which A has repeated channel actions.

$$\begin{array}{l} *[\ A?x; \\ \quad [x > 10 \longrightarrow A?y [] \mathit{else} \longrightarrow y := x + 3]; \\ \quad \underline{[y > 10 \longrightarrow A?z [] \mathit{else} \longrightarrow z := 0];} \\ \quad B!(x + y + z)\ ] \end{array}$$

This program can be re-written to eliminate repeated channel actions in a number of different ways. By way of illustration, we introduce a 2-bit state variable s and manually rewrite this program so that all operations on A are grouped into a single selection statement. The rewritten program is shown below. This process is equivalent to the original CHP process under the behavior set semantics used by slack elasticity.

$$\begin{array}{l} s := 0; \\ *[\ [s = 0 \longrightarrow A?x [] s = 1 \longrightarrow A?y [] s = 2 \longrightarrow A?z]; \\ \quad [s = 0 \longrightarrow \\ \quad\quad [x > 10 \longrightarrow s := 1 \\ \quad\quad [] \mathit{else} \longrightarrow y := x + 3; \\ \quad\quad\quad\quad \underline{[y > 10 \longrightarrow s := 2 [] \mathit{else} \longrightarrow z := 0]} \\ \quad\quad] \end{array}$$

[1]A brief summary of the CHP syntax is provided in the appendix.

$\talloblong s = 1 \longrightarrow$
$\underline{[y > 10 \longrightarrow s := 2 \talloblong else \longrightarrow z := 0; s := 0]}$
$\talloblong s = 2 \longrightarrow s := 0$
];
$[s = 0 \longrightarrow B!(x + y + z) \talloblong else \longrightarrow skip]$
]

This type of manual rewriting can be used to eliminate multiple channel actions, permitting the translation of the original process into dataflow components using DDD/STF. However, the control structure of the rewritten program can become quite complex, and the circuit logic can become redundant, as highlighted by the underlined statements, especially as the number of repeated channel actions increases.

To address this issue, this paper presents a systematic approach for directly supporting multiple channel actions during circuit synthesis, utilizing dataflow components that are always slack-elastic. This choice means that the final dataflow graph that results from a combination of our multiple channel access solution and DDD/STF is also entirely slack elastic, permitting any section of the graph to be pipelined without correctness concerns. Our approach exploits the original computation structure to dynamically compute the precise access sequence for the repeated channel. This information is used to generate control tokens for steering data and receiving results to and from the channel in the correct order. Through analysis and simulation, we show that our approach is well-suited for high-throughput asynchronous dataflow circuit synthesis in comparison with existing approaches.

The rest of the paper is organized as follows. We provide a summary of the dataflow computation model, identifying the subset that guarantees slack elasticity (Section II). We describe use cases for repeated channel actions including a related problem of sharing hardware units, and show how existing solutions can be adapted to address the repeated channel access problem (Section III). Next, we present our approach (Section IV), and quantify the differences between our proposed method and our adaptation of existing approaches (Section V).

II. Dataflow circuits

Dataflow circuits consist of concurrent, independent dataflow elements. Each dataflow element communicates with the predecessors and successors through local handshake protocols; thus it does not require global control. A dataflow element remains idle until it receives input tokens from its predecessors [11]. There are eight commonly used dataflow elements shown in Fig. 1. We provide CHP descriptions for each of them, using the channel names from Fig. 1.

1) $\textit{MERGE} \equiv$
   $*[c?b; [b = 0 \longrightarrow in1?x \talloblong b = 1 \longrightarrow in2?x]; out!x]$
2) $\textit{SPLIT} \equiv$
   $*[in?x, c?b; [b = 0 \longrightarrow out1!x \talloblong b = 1 \longrightarrow out2!x]]$
3) $\textit{COPY} \equiv$
   $*[in?x; out1!x, out2!x]$
4) $\textit{FUNC} \equiv$
   $*[in1?x, in2?y; out!f(x, y)]$

In general the $\textit{FUNC}$ block can have an arbitrary number of input channels; the example above is for a two-input function.

5) $\textit{SOURCE} \equiv *[out!const]$
6) $\textit{SINK} \equiv *[in?x]$
7) $\textit{BUF} \equiv *[in?x; out!x]$
8) $\textit{INIT} \equiv out!const; *[in?x; out!x]$

where *const* is a constant. All of these dataflow elements are deterministic, and can be written as CHP processes where the guards in selection statements only use local variables. This is sufficient to ensure that dataflow circuits constructed exclusively from these elements are always guaranteed to be slack elastic [1]. Static token form (STF) translates CHP programs into these eight dataflow circuits, and we use STF for the remainder of this paper when discussing dataflow circuit synthesis. In addition, the $\textit{MIXER}$ is a ninth dataflow element:

9) $\textit{MIXER} \equiv$
   $*[[|\overline{in1} \longrightarrow in1?x; b := 0 \talloblong \overline{in2} \longrightarrow in2?x; b := 1|];$
   $out!x, cOut!b$
   ]

Note that the $\textit{MIXER}$ has been written as a *non-deterministic* selection statement (see appendix). If multiple input tokens arrive at the input ports to a $\textit{MIXER}$, both probes $\overline{in1}$ and $\overline{in2}$ can be true simultaneously, and in this case the output is non-deterministic. Often dataflow graphs that use $\textit{MIXER}$s impose a mutual exclusion constraint on input token arrival so as to preserve deterministic execution. However, in general, a dataflow graph containing a $\textit{MIXER}$ may or may not be slack elastic, depending on the context in which the $\textit{MIXER}$ element is used.

An example of a dataflow graph that uses a $\textit{MIXER}$ and that can result in non-deterministic execution is shown in the dataflow circuit in Fig. 2. Input tokens arriving on channel A are either routed to the top function block (that adds one) or the

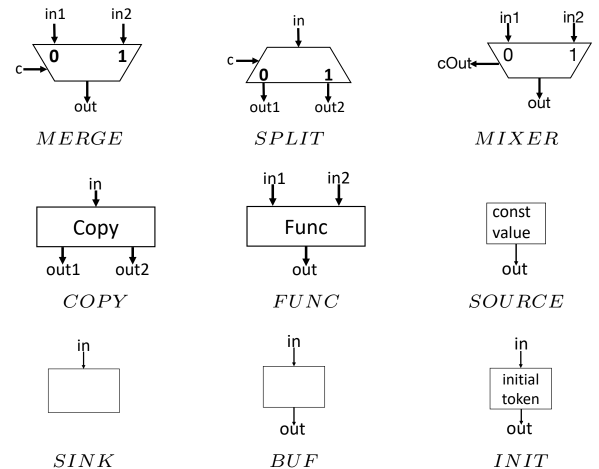


Fig. 1: Dataflow elements. Elements $\textit{MERGE}$, $\textit{MIXER}$, and $\textit{FUNC}$ can have more than two inputs (in*X* channels), and elements $\textit{SPLIT}$ and $\textit{COPY}$ can have more than two outputs (out*X* channels).

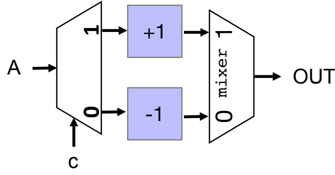


Fig. 2: A dataflow graph involving a mixer that can exhibit non-deterministic behavior.

bottom function block (that subtracts one), depending on the control specified on c. The result of the function operation is mixed to channel OUT. Now if only one token can be between A and OUT, then the mixer behaves in a deterministic manner, and the sequence of tokens on OUT is deterministic. However, if multiple tokens can be between A and OUT—in particular a token in both the increment function block and decrement function block—then the output can be non-deterministic. Note that Corollary 2 from [1] proves that, if we construct our dataflow graphs only using elements (1)–(8), such non-deterministic behavior cannot occur no matter how many tokens are in flight in the dataflow circuit.

III. Use Cases for Repeated Channel Actions

Repeated channel access naturally arises when designing circuit using CHP, such as in the case of a variable-length packet router. The receiving channel is initially used to receive the packet header. Based on the length information encoded in the header, the same channel is then iteratively accessed to receive the remaining packet stream.

Another case where repeated channel actions arise is when complex hardware units are shared. For example, consider a program that accesses some complex state machine from two different program points:

$$s := 0;$$
$$*[..; r := f(a, s); s := n(a, s); ..; t := f(b, s); s := n(b, s)]$$

If functions $f(\cdot)$ and $n(\cdot)$ are complex and rarely used, it is natural to factor them out into their own hardware component as follows rather than having two copies of the functions in the hardware implementation. This transformation corresponds to introducing a process that encapsulates the functions:

$$s := 0;\ \ *[\ IN?x; OUT!f(x, s); s := n(x, s)\ ]$$

The original process is modified to access this shared hardware resource as follows:

$$*[..;\ \ IN!a; OUT?r;\ \ ..;\ \ IN!b; OUT?t;\ \ ..\ ]$$

In this example, IN and OUT both have repeated channel actions. This transformation was used, for example, in a GPS baseband processor to share the complex numerically controlled oscillator logic across six channels in the GPS tracking loop and reduce the overall area of the design [12].

The repeated channel actions on IN and OUT mean that this process cannot be translated directly into a dataflow circuit using STF. To enable dataflow circuit generation, we introduce fresh channel replicas IN_i/OUT_i for each occurrence of IN and OUT in the original program. The resulting CHP without repeated channel actions is shown below:

$$*[..;\ \ IN_0!a; OUT_0?r;\ \ ..;\ \ IN_1!b; OUT_1?t;\ \ ..\ ]$$

This CHP conforms to the requirements of STF, and hence can be directly translated into a dataflow graph. What remains is the problem of combining the operations on the introduced replicas IN_i (OUT_i) to reconstruct the original channel operations on IN (OUT).

A. Standard solution without internal state

If the repeated channel access is a result of the introduction of shared hardware, and additionally if the shared hardware does not contain internal state (i.e. implements a pure function), then there is a straightforward solution to this problem: (i) Introduce a $\textsc{mixer}$ to combine all the freshly created IN_i channels into an IN channel; and (ii) Combine all the OUT_i channels using a $\textsc{split}$, whose control input is generated by the $\textsc{mixer}$, and where the data input from the $\textsc{split}$ is the output from the shared hardware.

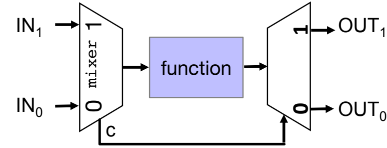


Fig. 3: Non-deterministic multiplexing of a function block.

This approach is illustrated in Fig. 3. This is a well-known solution to sharing a common function computation with non-deterministic multiplexing; for example, it was used in [12]. Unfortunately, this solution only applies to the special case of shared functions without internal state. Since the $\textsc{mixer}$ is permitted to re-order access to the shared function, state changes may not commute—leading to incorrect results.

In particular, the original problem statement where we had a repeated channel action cannot be handled with this approach, since re-ordering the sequence of token values communicated on primary output channels is normally an error.

For the rest of this section, we continue to examine the shared hardware use case, but focus on the situation where there is internal state in the shared hardware.

B. Adapting an existing approach: mutual exclusion

If the environment of Fig. 3 can *guarantee* that the $\textsc{mixer}$ input is mutually exclusive, and also that the $\textsc{mixer}$ input requests occur in the original CHP program order, then Fig. 3 is a valid approach to sharing the function hardware.

A further improved approach shown in Fig. 4 separates the control circuit and datapath. In this approach, the control channels cIN_i which corresponds to the data channels IN_i must still fulfill the two requirements mentioned above. The control token generated by the $\textsc{mixer}$ is used for controlling $\textsc{split}/\textsc{merge}$ in the datapath to deliver data tokens [13].

We can adapt this idea for the repeated channel access problem as follows.

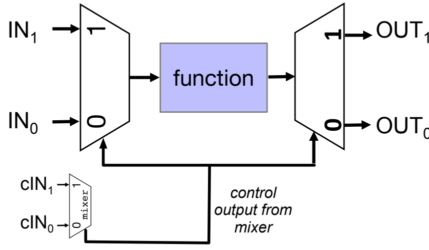


Fig. 4: Control/Data separation for shared unit access.

a) Output channels: For output channel O, we create multiple fresh channel replicas O_i for each location, and use a $\textit{MIXER}$ circuit directly (similar to Fig. 3) to combine them into the single O channel on which the final output is produced. This is analogous to the IN channel in Fig. 3.

b) Input channels: For input channel I, we can use a similar idea. We create multiple fresh I_i channel replicas, one for each location of the repeated input channel I. We introduce a new dataflow element, $\textit{UNMIXER}$, which has the following CHP behavior:

$$*[I?x;\ [\,|\overline{I_0} \longrightarrow I_0!x \,[]\, \overline{I_1} \longrightarrow I_1!x\ \ldots [] \,\overline{I_{n-1}} \longrightarrow I_{n-1}!x\,|\,]\,]$$

Once again, we must ensure that the O_i/I_i channels are mutually exclusive, and requests for output/input occur in the original CHP program order. It should be clear that, in both the input and output channel scenarios, we can also apply the control/data separation idea illustrated in Fig. 4.

The use of channel probes (in both $\textit{MIXER}$ and $\textit{UNMIXER}$) means that the final dataflow circuit is not guaranteed to be slack elastic by construction, because probes are used in the guards of selection statements [1]. In particular, since we require mutual exclusion on the inputs/outputs of the $\textit{MIXER}/\textit{UNMIXER}$, we cannot add slack on the channels directly connected to their data ports—limiting pipelining choices. We will show how this can degrade overall performance (Section V).

IV. A Slack Elastic Approach

Our proposed solution to the repeated channel access problem is to use an entirely slack elastic approach. In other words, our design is dataflow circuit that does not use either $\textit{MIXER}$s or $\textit{UNMIXER}$s; it only uses elements (1)–(8) from Section II. Given that STF can introduce concurrent accesses in the presence of repeated channel actions, we can use the same starting point as in Section III-A—namely, for each repeated channel action, we introduce a channel replica. However, rather than using a $\textit{MIXER}$ for output ports, we use a $\textit{MERGE}$; similarly, we use a $\textit{SPLIT}$ instead of an $\textit{UNMIXER}$ for input ports. In both cases, what remains is the dataflow circuit that generates the correct control token for the $\textit{MERGE}/\textit{SPLIT}$ that corresponds to the order in which the repeated channel action occurred in the original CHP process. We call the control channel $\textit{CTRL}$ in what follows.

To illustrate our approach, consider the alternating split example $*[L?x; A!x; L?x; B!x]$. We create two L channel replicas, and rewrite the process as $*[L_0?x; A!x; L_1?x; B!x]$. To control the $\textit{SPLIT}$ that delivers data tokens from L to L_0 and L_1, the $\textit{CTRL}$ sequence is $\{0, 1, 0, 1, \ldots\}$. The process generating this sequence is: $s := 0; *[C!s; s := 1-s]$, where C generates the $\textit{CTRL}$ sequence. This process itself satisfies STF constraints and can be readily mapped to a dataflow circuit. In the general case, the sequence of control tokens can be data-dependent (e.g. the router example in Section III), and cannot be generated so easily. In what follows, we provide an elegant method to directly generate the $\textit{CTRL}$ sequence by structural induction on the original CHP program.

CHP processes have three types of control structures: sequential, conditional and iterative. We remark that recent work has shown how more complex control structures can be automatically modified to correspond to conditional/iterative structures [14]; hence this approach generalizes to more complex control structures as well. For clarity, we show how our approach can be used when the behavioral description is in the CHP syntax.

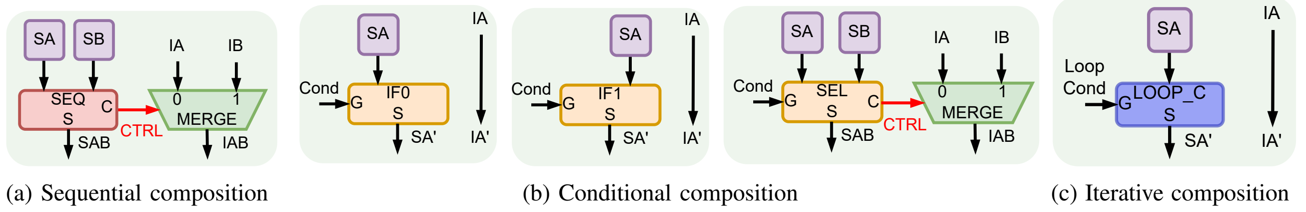
For each repeated channel action in the original program, we introduce a base channel replica. For each type of control structure that involves channel action, as shown in Fig. 5, a new inductive channel replica is introduced, which can either be directly connected to the existing channel replica or formed by combining two channel replicas using a $\textit{MERGE}$ (for output channels) or a $\textit{SPLIT}$ (for input channels).

For each channel replica, we introduce a new auxiliary one-bit *Access Sequence* that encodes how many times the channel replica is used at run-time via run-length encoding, shown in purple blocks in Fig. 5. The *Access Sequence* starts with N '*1*' tokens, where N is the number of the times the channel replica is accessed (N can be zero), followed by one '*0*' token indicating the end of usage in one iteration of the program fragment. The *Access Sequence* is computed at run-time and used to compute the $\textit{CTRL}$ token sequence for $\textit{MERGE}/\textit{SPLIT}$, which is shown as red arrows in Fig. 5.

Given a specific channel in a program fragment, we will show how we can construct an inductive channel replica by using $\textit{MERGE}/\textit{SPLIT}$, and compute the $\textit{CTRL}$ token sequence for $\textit{MERGE}/\textit{SPLIT}$. We also calculate a new accompanying *Access Sequence* that correctly counts the number of times the inductive channel replica is accessed during the dynamic execution of that program fragment.

To better illustrate our method, take the following program as an example. A has repeated channel actions and is already replaced by the base channel replicas.

```
res := 0;
*[ A0?a0;
   *[a0 < 10 ⟶ B?b0;
      [b0 = 0 ⟶ A1?a1; res := res + a1
      [] else ⟶ A2?a2; A3?a3; res := res + a2 + a3
      ];  a0 := a0 + 1
   ];
   RES!res
 ]
```



(a) Sequential composition (b) Conditional composition (c) Iterative composition

Fig. 5: Control processes for three control structures

a) Base Case: If the base channel replica is used once in a statement, then the *Access Sequence* is repeated sequence of *1* followed by *0*: {*1, 0, 1, 0, ...*}. The process generating this sequence is $s := 1; *[S!s; s := 1 - s]$ where S sends the *Access Sequence*. This can be directly built with deterministic dataflow elements without any difficulty. If the statement does not use the channel (e.g. it is an assignment, skip, or a communication action on a different channel), there is no channel replica and no accompanying *Access Sequence*.

In our example, the base channel replicas A_0, A_1, A_2, A_3 has the same {*1, 0, 1, 0, ...*} repeated *Access Sequence* respectively, shown as purple blocks in STEP 1 in Fig. 6.

b) Induction: The hypothesis used for induction is that every program fragment either has one inductive channel replica (potentially formed by using MERGE/SPLIT) with an associated *Access Sequence*, or no channel replica and no associated *Access Sequence*.

Sequential composition: $A; B$: if the two program fragments A and B both contain access to the channel of interest, then we know that these accesses must be ordered by servicing those from A first followed by those from B. This is handled as follows: if neither fragment or only one fragment has channel replica, there is nothing to do; otherwise, we generate CTRL token sequence, which is used to control combining two channel replicas into a new one, as well as generate a new associated *Access Sequence*. The process *SEQ* is shown as below and illustrated in Fig. 5(a).

$$\begin{aligned} SEQ \equiv s\downarrow; *[&[\neg s \longrightarrow SA?seq \;[]\; s \longrightarrow SB?seq]; \\ &[seq \longrightarrow C!s \;[]\; \neg seq \longrightarrow s := \neg s]; \\ &[s \wedge \neg seq \longrightarrow skip \;[]\; else \longrightarrow S!seq] \\ ]& \end{aligned}$$

The process combines IA and IB to form a new inductive channel replica IAB by a MERGE, which is controlled by the CTRL token generated from the *SEQ* circuit block. In the *SEQ*, channel *SA* receives *Access Sequence* of channel replica *IA* in the program fragment *A*, channel *SB* receives that of *B*. It uses the information to calculate the CTRL tokens sent by C, as well as the new associated *Access Sequence* for IAB sent by S. The process uses s to keep track of which program fragment is being processed: A or B. The *seq* is used to determine when to switch to the other program fragment.

For the example, consider the sequential fragment $A_2?a2; A_3?a3$. The corresponding partial control circuit is shown in STEP 2 in Fig. 6. The new inductive channel replica A_{23} is formed by combining A_2 and A_3 using SPLIT, as A is an input channel. The CTRL sequence $C0$ is repeated {*0, 1, 0, 1,...*} generated by the *SEQ*. It controls the SPLIT to deliver data tokens from A_{23} to A_2 and A_3 alternatively. The *Access Sequence* corresponds to the new channel replica A_{23}, $S0$, is repeated {*1, 1, 0, 1, 1, 0,...*}. If this sequential fragment is executed, A_{23} will be accessed consecutively twice.

Two-way selections: $[guard = 0 \rightarrow A [] else \rightarrow B]$. If neither fragment has channel replica, there is nothing to do. If there is only one channel replica, then it will be directly connected to the new channel replica, so we don't need to generate CTRL token to combining any replicas. However, the *Access Sequence* is modified as the channel is used conditionally. The process used to generate the new *Access Sequence* is shown below and illustrated in the left and middle of Fig. 5(b).

$$\begin{aligned} IF0 \equiv\ & seq\downarrow; \\ & *[[\neg seq \longrightarrow G?g [] else \longrightarrow skip]; \\ & \quad [g = 0 \longrightarrow SA?seq [] else \longrightarrow seq := 0]; S!seq \\ &] \end{aligned}$$

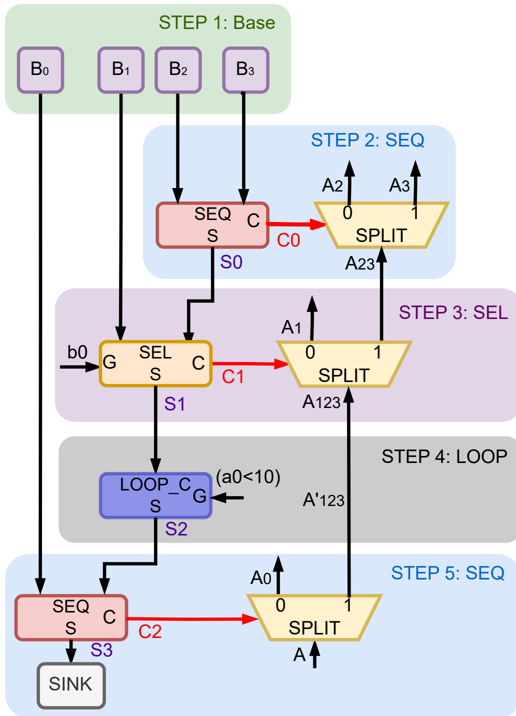


Fig. 6: Control circuit constructed inductively based on the program

$$\begin{aligned}&IF1 \equiv seq{\downarrow};\\ &\quad *[[\neg seq \longrightarrow G?g \,[]\, else \longrightarrow skip];\\ &\qquad [g=0 \longrightarrow seq := 0 \,[]\, else \longrightarrow SB?seq];S!seq\\ &\quad]\end{aligned}$$

IF0 is used when A has channel replica, while IF1 is for the other case. Channel G receives condition value and *SA'* is the new *Access Sequence* for IA'. In both cases, we require the value of the guard (an input on channel G) from the original program. This is done by modifying the original program to: "$G!guard$; [$guard = 0 \rightarrow A \,[]\, else \rightarrow B$]." The processes either generate a single *0* if the channel is not used, or they simply propagate the *Access Sequence* to the output if the channel is used.

If both fragments have channel replicas, the process of generating new *Access Sequence* and CTRL token sequence is shown as follows and the illustration is shown in the right of Fig. 5(b).

$$\begin{aligned}&SEL \equiv seq{\downarrow};\\ &\quad *[[\neg seq \longrightarrow G?g \,[]\, else \longrightarrow skip];\\ &\qquad [g=0 \longrightarrow SA?seq \,[]\, else \longrightarrow SB?seq];\\ &\qquad [seq \longrightarrow C!g \,[]\, else \longrightarrow skip];S!seq\\ &\quad]\end{aligned}$$

As in the previous case, the original program has to send the value of the guard via the channel G.
Consider the selection fragment in the example:
[$b0 = 0 \rightarrow A1?a1 \,[]\, else \rightarrow A2?a2; A3?a3$].
As shown in STEP 3 in Fig. 6, the partial control circuit is inductively built upon the previous one. The base channel replica A_1 and the inductive channel replica A_{23} are combined to a new inductive channel replica A_{123}. The *Access Sequence* for A_1 and A_{23} are input into the *SEL* circuit block. According to the guard value $b0$ sent by the original program, the CTRL $C1$ and *Access Sequence* $S1$ accompanying the new inductive channel replica A_{123} are generated.

Assume that the $b0$ sequence is {*0, 1, 1*}, then the $S1$ token sequence is $\{\underbrace{1,0}_{\text{from B1}}, \underbrace{1,1,0}_{\text{from S0}}, \underbrace{1,1,0}_{\text{from S0}}\}$. And the $C1$ sequence is { *0, 1, 1, 1, 1* }, since A_1 is accessed once, A_2, A_3 are accessed alternatively twice, A_{23} is accessed four times in total.

Loops: *[$guard \rightarrow A$], where the guard value could be updated in the program fragment *A*. If fragment *A* doesn't have a channel replica, we're done; otherwise, we use the following process, shown in Fig. 5(c) to generate new *Access Sequence* and we don't need the CTRL token, since *IA* is directly connected to the new channel replica IA' for iterative fragment.

$$\begin{aligned}&LOOP_C \equiv\\ &\quad seq{\downarrow},\ \ g{\downarrow};\\ &\quad *[[\neg seq \longrightarrow G?g \,[]\, else \longrightarrow skip];\\ &\qquad [seq \vee g \longrightarrow SA?seq \,[]\, else \longrightarrow skip];\\ &\qquad [(g \wedge seq) \vee (\neg g \wedge \neg seq) \longrightarrow S!seq \,[]\, else \longrightarrow skip]\\ &\quad]\end{aligned}$$

Like condition structure, the original program run in parallel with LOOP_C should send the loop condition value, and is re-written as follows: $G!guard$;*[$guard \rightarrow A$; $G!guard$].

But in this re-written process, the channel *G* is used at two different points in the program again, so we further decompose this process to the following two processes:

$$\begin{aligned}&\quad G0!guard;*[guard \longrightarrow A;\ \ G1!guard]\\ &\| \quad s{\downarrow};*[[\neg s \longrightarrow G0?s \ \ [] s \longrightarrow \ \ G1?s];G!s]\end{aligned}$$

As for the example, the control circuit generated for the loop fragment is shown in STEP 4 in Fig. 6. The new inductive channel replica A'_{123} is directly connected to A_{123}. And the corresponding *Access Sequence* $S2$ is generated according to the loop guard value sent from the original program. Assume that the $b0$ token sequence is the same as above, and that the loop is executed three times. The $S2$ token sequence is
$\{\underbrace{1}_{\text{from B1}}, \underbrace{1,1}_{\text{from S0}}, \underbrace{1,1}_{\text{from S0}}, 0\}$
Bascially, the $LOOP_C$ eliminates the *0* tokens in the $S1$ sequence and adds an addition *0* at the end to indicate the completion of using channel replica A'_{123}.

Note that every circuit block we have introduced conforms to the requirements of STF for dataflow synthesis. Hence, they can all be translated to MIXER-free dataflow components using a standard dataflow synthesis approach.

After applying these transformations inductively, all the channel replicas are combined into one channel which is connected to input/output port. The CTRL tokens, used to guarantee mutual exclusive and in-order accesses, are generated by the dataflow circuit constructed by structural induction.

As for the example, there is one last sequential composition, and the corresponding partial control circuit is shown in STEP 5 in Fig. 6. The base channel replica A_0 and the inductive channel replica A'_{123} are combined using SPLIT, with the new channel replica being connected to channel A in the environment. The *Access Sequence* for channel A is not used anymore, and can be absorbed by the *SINK* dataflow element.

V. Performance Analysis and Simulation Results

A. Simulation Setup

All the examples are written using the open-source ACT dataflow language [15], and synthesized to dataflow circuits by dflowmap [16]. The synthesized dataflow circuit are composed of standard dataflow building blocks and application-specific computation units. The delay information of standard dataflow building blocks are collected by running SPICE simulation, and the delay of computation units are generated by Cadence's logic synthesis tool. We use actsim [17], which takes delay information as annotations, to run the simulation.

B. Shared Unit with Internal State

As explained in III-A, in the standard solution, mutual exclusion is not guaranteed at the input ports of the MIXER. So the shared unit may be accessed out of program order, resulting incorrect behavior if the shared unit has internal state.

Take the following process as an example, where $FMADD$ is factored out as a shared unit.

$EXAMPLE_1 \equiv$
*[...;
  $c0 := FMADD(a0, b0)$;
  $c1 := FMADD(a1, b1)$;
  $c2 := FMADD(a2, b2)$;
  $res := c0 + c1 + c2$;
  ...
]

$FMADD$ has internal state characterized by the $count$ and sum local variables.

$FMADD \equiv$
  $count := 0$; $sum := 0$;
  *[$I0?a$, $I1?b$;
    $sum := a*b + sum$; $O!sum$;
    [$count = 2 \longrightarrow sum := 0,\ count := 0$
    [] $else \longrightarrow count := count + 1$
    ]
  ]

The value of local variable sum can differ depending on the order the shared unit is accessed. In our approach, when the $FMADD$ unit takes $a0$ and $b0$ as inputs, the sum is always already reset to 0; this is not guaranteed by the standard solution (Sec. III-A). For completeness, we simulated both options and confirmed that only the circuit that uses the standard solution produces erroneous results.

C. Comparison with Mutual Exclusion Approach

The approach in [13] adapts mutual exclusion method outlined in III-B. It applies datapath and control decoupling, so the circuit for combining channel replicas is the same as that shown in Fig. 4. This approach leverages the existing in-order control circuit used for controlling datapath, and allows only one token flowing through the control network. This token is used to track the active fragment as program executes, and trigger sending a cIN_i token to the $\textsc{mixer}$. In this way, the control circuit can guarantee the $\textsc{mixer}$ input is mutually exclusive, and also that the $\textsc{mixer}$ input requests occur in the original program order. We implement $EXAMPLE_1$ using this method and the generated circuit exhibits slightly better throughput than ours. This is because our control circuit introduces a small overhead compared to [13] in this example which has the simplest possible control (only sequential compositions). However, our approach is superior in more general scenarios, for the two reasons presented below.

(1) The control circuit in [13] strictly follows the control flow of the original program, resulting in the control token flowing through unnecessary control logic. In contrast, our approach directly skips the control structures that do not involve repeated channel actions. In the following process as an example, the $FMADD$ unit is factored out as a shared unit.

$EXAMPLE_2 \equiv$
*[...
  [$g0 \longrightarrow c0 := FMADD(a0, b0)$ [] $else \longrightarrow c0 := a0$];
  [$g1 \longrightarrow c1 := c0$ [] $else \longrightarrow c1 := 2*c0$];
  [$g2 \longrightarrow c2 := FMADD(c1, c1)$ [] $else \longrightarrow c2 := c1$];
  ...
]

The control circuit used in approach [13] includes three conditional control structures, with the second one being unnecessary for the shared channel access problem, as there is no channel action in the second condition statement. In this case, the token flowing in the control circuit can be stalled by unnecessarily waiting for the guard value $g1$, which prevents the new token from entering the control circuit, since only one token can be present in the control circuit to guarantee mutual exclusion. So even the second location doesn't access the shared unit because $g2$ is false, the first location cannot use the shared unit due to the lack of the control token.

In contrast, in our approach, the second conditional statement will be directly skipped. Coordinating the access to the shared unit depends only on the guard value $g0$ and $g2$. As soon as it is known that the second location does not access the shared unit, the data from the first location can be fed into the shared unit if $g0$ is true. Generally speaking, our approach grants access to the shared unit at the earliest correct time.

To validate this analysis, we simulated both versions. Our approach achieves **1.85x** better throughput than the approach [13]. If the number of control structures between two sharing locations increases, the shared unit may remain unnecessarily idle for a longer period in the approach [13], further degrading performance.

(2) The approach [13] uses $\textsc{mixer}$ in the control circuit to manage data token delivery. Here, we show the dataflow circuit for a conditional statement.

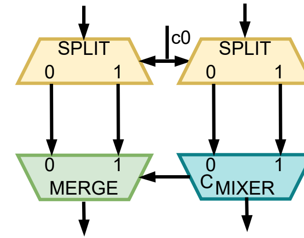


Fig. 7: Dataflow circuit for a conditional statement in approach [13]

As shown in Fig. 7, the control token generated by the $\textsc{mixer}$ in the control circuit is used to control $\textsc{merge}$ in the datapath. In this case, the data tokens cannot be delivered to the next computation stage by $\textsc{merge}$ until the corresponding control structure owns the control token. However, to guarantee mutual exclusion, only one token can be present in the control circuit, which might eliminates some potentials of parallel execution. Consider the example shown as below, the $FMADD$ unit is factored out as a shared unit.

$EXAMPLE_3 \equiv$

```
*[ ...
  [g0 ⟶ c0 := FMADD(a0, b0) [] else ⟶ c0 := a0];
  [g1 ⟶ *[g2 ⟶ c1 := FMADD(c0, c0); ...]
  [] else ⟶ c1 := FUNC(c0)
  ];
  ... ]
```

Assume both the $g0$ and $g1$ token sequence are $\{1, 0\}$. In the first iteration of execution, the $*[g2 \rightarrow \ldots]$ loop statement is active, and the shared unit *FMADD* is accessed iteratively. Meanwhile, the $a0$, $b0$ tokens are ready for the next iteration of execution. Ideally, since $g0$, $g1$ are both false, the data tokens can be delivered to the *FUNC* unit. In this scenario, *FMADD* and *FUNC* can execute in parallel.

However, in the approach [13], if the token is used in the $*[g2 \rightarrow \ldots]$ loop control structure, even though the $c0$ token is ready, it cannot be delivered to the *FUNC* block. This is because the MERGE in the datapath waits for the flowing token to exit the loop control structure and re-enter the control circuit, thereby preventing parallel execution of the *FMADD* and *FUNC* unit. Our approach doesn't use MIXER in either the control circuit or the circuit for combining channel replicas, so there is no token number constraint in our control circuit. Through simulation[2], we demonstrate that our approach enables the parallel execution of the *FMADD* and *FUNC* unit, achieving **1.35x** better throughput than approach [13]. The simulation results are summarized in TABLE I.

TABLE I: Simulation results

	Execution Time (ns)		Ratio
	Ours	Approach [13]	
EXAMPLE_1	70716	**67142**	**0.95**
EXAMPLE_2	**2205411**	4073044	**1.85**
EXAMPLE_3	**5422811**	7335725	**1.35**

D. Control Circuit Throughput and Optimization Results

Recent work [18] improves the approach in [13] by resolving the two limitations outlined in V-C. For a specific channel, it constructs the smallest necessary subset of the original in-order control circuit and directly calculates the control token for the SPLIT/MERGE in the datapath, rather than using the MIXER to generate it. However, the MIXER is still used in a non-slack elastic manner in the circuit for combining channel replicas, as shown in Fig. 4. This has two drawbacks: (1) scalability—it limits circuit throughput with increased sharing, and (2) renders optimization techniques based on slack elasticity unusable, thereby lowering throughput.

1) Scalability: To support high-throughput asynchronous dataflow circuit synthesis, we typically limit the number of inputs of MERGE to four. A high fan-in MERGE is constructed using a tree of smaller MERGE units. The same strategy can be applied to constructing multi-way MIXER. However, having multiple tokens in the MIXER's tree structure as used by [13], [18] leads to non-deterministic behavior. To resolve this issue, the environment must ensure that no new input request is sent until the multi-way MIXER's output is generated; i.e. the control circuit from [13], [18] cannot send the cIN_i token until it receives notification that the MIXER has produced its output. This path can become the critical path in the circuit as the number of repeated channel actions increases.

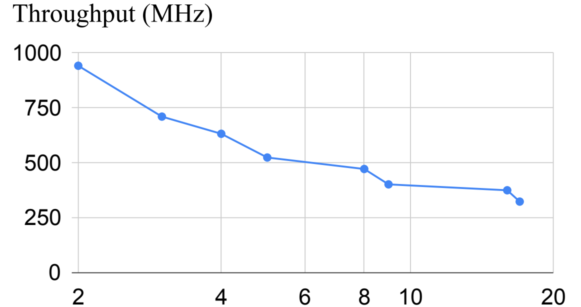


Fig. 8: Control token throughput in approach [18]

We quantifed the control token throughput in [18] by constructing a tree-structured MIXER and a tree-structured notification circuit using SPLIT in a 28nm technology. The simulation results, shown in Fig. 8, show that the throughput decreases as the number of repeated channel actions increases. When the number reaches 5, the control token throughput is degraded to approximately 500 MHz. Even if the MIXER is optimized, the latency of the element will increase rapidly with the number of inputs and degrade the throughput of the overall circuit in this control configuration.

In contrast, all the components in our approach are slack elastic. This allows the circuit to be pipelined enabling multiple tokens to be in flight as needed. Take the tree-structured MERGE as an example, by pipelining, the throughput of it is the same as a single two-way MERGE. Our approach is well-suited for high-throughput asynchronous dataflow synthesis, particularly for circuits with shared units (e.g. memories) that can be accessed from many locations in the program.

2) Enabling optimizations: Table II (from [14]) summarizes performance improvements achieved through dataflow optimizations that require slack elasticity. These optimizations cannot be applied to circuits utilizing the MIXER in a non-slack elastic manner, as in [18]. The optimized circuits achieve a **1.74x** improvement in latency and a **1.57x** improvement in throughput on average. Our approach can be combined with these optimizations, unlike [13], [18].

E. Automation

We have implemented the structured construction of control outlined above in the ACT EDA flow for asynchronous logic. In particular, we can automatically translate arbitrary probe-free CHP programs into slack-elastic dataflow graphs. The implementation follows the recursive construction described in Section IV.

[2]In order to illustrate our analysis by simulation, we select the *FUNC*'s delay to ensure it is on the critical path of the circuit.

TABLE II: Improvements from applying optimization techniques based on slack elasticity

	Delay (ps)		Throughput (MHz)	
Benchmark	**Original**	**Optimized**	**Original**	**Optimized**
adpcm-u	11104	4588	367	560
dfadd-a	9445	7415	212	278
differential	226516	165814	4	6
gsm-d	41378	26861	25	39
mpeng-d	15844	7523	99	193

VI. Conclusion

We presented a systematic approach to the repeated channel actions problem in dataflow circuit synthesis, utilizing dataflow elements that exhibit deterministic behavior at all times. To the best of our knowledge, this is the first approach that enables the generation of slack elastic dataflow circuits with general resource sharing capabilities. By generating the *Access Sequence* inductively at run-time, the distributed control logic tracks the number of times the shared channel is accessed in a program fragment. We dynamically generate a $CTRL$ token sequence based on *Access Sequence* to keep the correct order of accesses from multiple locations. We demonstrated that our approach is well-suited for high-throughput asynchronous dataflow circuit synthesis, providing significant throughput improvement compared to existing approaches.

Appendix

The notation we use is based on Hoare's CSP [19]. A full description of the notation and its semantics can be found in [20]. What follows is a brief description of the notation.

- **Skip**: $skip$ does nothing.
- **Assignment**: $a := b$. This statement means "assign the value of b to a." We also write $a\uparrow$ for $a := true$, and $a\downarrow$ for $a := false$.
- **Deterministic Selection**: $[G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n]$ where G_is are boolean expressions (guards) and S_is are program parts. The execution of this command corresponds to waiting until one of the guards is true, and then executing one of the statements with a true guard. The guards must be stable and mutually exclusive. The notation $[G]$ is shorthand for $[G \rightarrow skip]$, which corresponds to waiting for G to become true.
- **Non-deterministic Selection**: $[| G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n |]$ resembles the deterministic selection, but more than one guard G_i can be true. Other than this, the behavior is the same as the deterministic selection.
- **Repetition**: $*[G \rightarrow S]$. Executes S while G is true.
- **Send**: $X!e$ means send the value of e over channel X.
- **Receive**: $Y?v$ means receive a value over channel Y and store it in variable v.
- **Probe**: The boolean $\overline{X}$ is true if and only if a communication over port X can complete without suspending.
- **Sequential Composition**: $S; T$
- **Parallel Composition**: $S \| T$ or S, T

References

[1] R. Manohar and A. J. Martin, "Slack elasticity in concurrent computing," in *Mathematics of Program Construction* (J. Jeuring, ed.), (Berlin, Heidelberg), pp. 272–285, Springer Berlin Heidelberg, 1998.

[2] A. J. Martin, "Programming in vlsi: From communicating processes to delay-insensitive circuits," tech. rep., Department of Computer Science, California Institute of Technology, 1989.

[3] I. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[4] I. Sutherland and S. Fairbanks, "Gasp: a minimal fifo control," in *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems*, pp. 46–53, 2001.

[5] M. Singh and S. M. Nowick, "Mousetrap: High-speed transition-signaling asynchronous pipelines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 684–698, 2007.

[6] S. M. Burns and A. J. Martin, "Syntax-directed translation of concurrent programs into self-timed circuits," in *Conference on Advanced Research in VLSI*, 1988.

[7] J. Kessels and A. Peeters, "The tangram framework: Asynchronous circuits for low power," in *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 255–260, 2001.

[8] C. Wong and A. Martin, "High-level synthesis of asynchronous systems by data-driven decomposition," in *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, pp. 508–513, 2003.

[9] J. Teifel and R. Manohar, "Static tokens: using dataflow to automate concurrent pipeline synthesis," in *10th International Symposium on Asynchronous Circuits and Systems*, pp. 17–27, 2004.

[10] R. Manohar, T.-K. Lee, and A. Martin, "Projection: a synthesis technique for concurrent systems," in *Proceedings. Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 125–134, 1999.

[11] J. B. Dennis, "The evolution of 'static' data-flow architecture," in *Advanced Topics in Data-Flow Computing*, Prentice-Hall, 1991.

[12] B. Z. Tang, S. Longfield, S. A. Bhave, and R. Manohar, "A low power asynchronous gps baseband processor," in *IEEE International Symposium on Asynchronous Circuits and Systems*, 2012.

[13] L. Josipović, A. Marmet, A. Guerrieri, and P. Ienne, "Resource sharing in dataflow circuits," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, pp. 1–9, 2022.

[14] R. Li, L. Berkley, Y. Yang, and R. Manohar, "Fluid: An asynchronous high-level synthesis tool for complex program structures," in *IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 1–8, IEEE, 2021.

[15] R. Manohar, "The dataflow sublanguage." Available at https://avlsi.csl.yale.edu/act/doku.php?id=language:langs:dflow (2023/02/11).

[16] R. Manohar, R. Li, Y. Yang, and O. Richter, "dflowmap." Available at https://github.com/asyncvlsi/dflowmap.

[17] R. Manohar, "Actsim: an act simulator." Available at https://avlsi.csl.yale.edu/act/doku.php?id=tools:actsim(2023/02/03).

[18] A. Elakhras, R. Sawhney, A. Guerrieri, L. Josipovic, and P. Ienne, "Straight to the queue: Fast load-store queue allocation in dataflow circuits," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 39–45, 2023.

[19] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[20] A. J. Martin, "Synthesis of asynchronous vlsi circuits," Tech. Rep. CS-TR-93-28, California Institute of Technology, 1993.