2025 29th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC) | 979-8-3315-0310-9/25/$31.00 ©2025 IEEE | DOI: 10.1109/ASYNC65240.2025.00022

Automated Decomposition of Concurrent Programs for Asynchronous Logic Synthesis

Karthi Srinivasan
Department of Electrical and Computer Engineering
Yale University
New Haven, USA
karthi.srinivasan@yale.edu

Rajit Manohar
Department of Electrical and Computer Engineering
Yale University
New Haven, USA
rajit.manohar@yale.edu

***Abstract*—Decomposition is the process of modifying a high-level description of an asynchronous circuit into an equivalent but more concurrent version with the goal of producing more performant circuits. We introduce a novel, general method for decomposing a system of abstract programs into an equivalent system with higher concurrency. The method uses control and data dependencies in the input program to determine independent portions and extract them into separate sub-processes. We formulate the problem of extracting these sub-processes in terms of finding cuts that satisfy certain constraints in a graph, and propose a heuristic to find these cuts that performs well. The decomposition algorithm can be repeatedly applied in an iterative manner to produce progressively more concurrent systems. The proposed procedure results in higher throughput at the cost of a more expensive circuit in terms of area and power consumption. A software tool that implements the decomposition algorithm is also demonstrated and benchmarked.**

***Index Terms*—Asynchronous Integrated Circuits, Concurrent Programs, Process Decomposition**

I. Introduction

The synthesis of asynchronous circuits involves transforming a high-level sequential specification into a system of circuits that implement the computation described. The characteristics of the final circuit that is generated depend on the design point that is targeted for that particular application. Techniques for synthesizing these circuits fall broadly into two categories. The first are syntax-directed translation methods [1, 2], which synthesize a given high-level program exactly as written, respecting all the synchronization behavior that is contained within it. While these are general purpose techniques, they lack the capacity to exploit concurrency that may be hidden in the sequential description of the computation. At the other extreme are dataflow-style techniques that break up the high-level program into smaller inter-connected processes running in parallel, each of which can be implemented using one of a small set of circuit templates (i.e. the dataflow elements). Pure dataflow synthesis can result in circuits that are over-pipelined and sacrifice latency, energy and area for an improvement in throughput [3]. The goal of our work is to develop a flexible tool that can introduce *some* concurrency, but where the user has a choice about the degree of decomposition introduced.

To ensure that we preserve the correctness of the original computation when automating decomposition, we restrict our attention to *slack elastic* programs [4]. Slack elasticity provides theoretical guarantees about the correctness of decomposition techniques that introduce concurrency, including pipelining and and projection [5]. Prior work in this area such as the static tokens method [6] results in extremely fine-grained pipelining, since the target physical architecture was an FPGA with a fixed set of dataflow primitives. The data-driven decomposition method [7] decomposes each variable in the original program into a separate process, each of which can be implemented by a single pre-charge half-buffer (PCHB) stage. This also leads to over-pipelining due to the resultant constraints on each process (e.g. all input communications must precede all output communications).

We present a general, automated technique to decompose sequential slack-elastic programs into a system of concurrent programs. The decomposition technique operates on an intermediate representation (IR) of the input program written in CHP (Communicating Hardware Processes), whose syntax is described in Appendix I. In standard compiler terminology, the decomposition is a transformation pass that is applied on the IR, prior to circuit synthesis; hence, it can be viewed as a CHP-to-CHP re-writing whose goal is to create a new program that improves performance.

We introduce a graph-based framework for analyzing decomposition opportunities in the CHP description of an asynchronous circuit that incorporates a combination of control- and data-dependencies, called the dependence-based decomposition graph. The framework allows us to formulate the problem of decomposition in terms of finding cuts in a graph that satisfy certain constraints. We also demonstrate a simple heuristic to find these cuts that is surprisingly efficient, and show that circuits synthesized from the decomposed CHP achieve superior throughput (1.4–13.3$\times$) compared to those synthesized from the original CHP, at the cost of higher power consumption, latency and area. Our decomposition process can be viewed as a generalization of previous work on fine-grained dataflow decomposition. In our approach, we permit the degree of pipelining/decomposition to be controlled, and no assumptions are made about the underlying circuit realization (e.g. dataflow elements only) of the behavioral description.

The rest of this paper is organized as follows. Section II introduces some background that forms the basis for this work. Section III describes our primary contribution, a general method for decomposition of concurrent message-passing programs. Section IV provides results from simulations of synthesized circuits, Section V details some immediate future work, and Section VI concludes the paper.

II. Background

We specify an asynchronous circuit as a system of concurrent processes, using the CHP notation. CHP processes communicate with each other via message-passing channels, which are used to send/receive values. Every channel in the system has a certain amount of slack, which is the maximum number of pending values that can exist on that channel at any given time. Slack-elastic programs are those whose correctness is unchanged when the slack on any channel is increased. Increasing slack on channels decreases synchronization across processes and can increase the concurrency in the system. Note that slack elasticity is a property of a closed system and hence the environment of a program must also be taken into account in order to make a determination. From prior work, it is known that when a closed system is deterministic and does not contain any channel probes, the system is slack elastic. A detailed analysis of slack elasticity can be found in [4].

Since increasing the slack on a channel can increase the number of execution traces, we use the sequence of values communicated over channels to specify the process [4]; the relative order of communication actions on different channels is not considered. This has the consequence of allowing several concurrent transformations. For example, consider the following simple CHP:

$$*[A?x; B?y; C!x; D!y]$$

Here, the communication actions on $\{A, C\}$ and $\{B, D\}$ are not data dependent and the ordering imposed by the semicolons is unnecessary. Under a slack elastic environment, this can be transformed into two concurrent buffers:

$$*[A?x; C!x] \quad \| \quad *[B?y; D!y]$$

The two systems have the same sequences of values sent over the channels and are hence equivalent.

An important transformation on the input CHP that enables analysis is the static token form (STF) [6]. The process of converting into STF consists of several transformation of the program. The first is to ensure that every variable is assigned exactly once. If a program consists of multiple assignments to the same variable, then a fresh variable is introduced to resolve this. For example, $*[A?x; x := x + 1; B!x]$ is converted to $*[A?x; y := x + 1; B!y]$. Next, we introduce ϕ- and ϕ^{-1}-functions, similar to those in static single information (SSI) form [8]. In SSI form, if a definition (write) of a variable at program point p reaches two uses (reads) at points p_1 and p_2, then either all paths (through the program) from p to p_1 contain p_2 or all paths from p to p_2 contain p_1. This constraint is typically violated at entry points of selections. The ϕ^{-1}-functions are used to resolve this and to capture the mapping of values for variables that were defined before a selection but used within it. Consider the following process:

$$\begin{aligned}&*[C?c, X?x;\\ &\quad [c = 0 \longrightarrow Y!x [] c = 1 \longrightarrow Z!x [] \mathit{else} \longrightarrow \mathit{skip}]\ ]\end{aligned}$$

Here, the value of the variable x has its value used in some of the branches of the selection and hence a ϕ^{-1}-function is introduced to obey the aforementioned rule:

$$\begin{aligned}&*[C?c, X?x; \{x_1, x_2, \bot\} := \phi^{-1}(x);\\ &\quad [c = 0 \longrightarrow Y!x_1 [] c = 1 \longrightarrow Z!x_2 [] \mathit{else} \longrightarrow \mathit{skip}]]\end{aligned}$$

In case the variable is unused in a branch, a dummy bottom/null variable ($\bot$) is placed as one of the outputs of the ϕ^{-1}-function. Note that the ϕ^{-1}-function has one input but several outputs, each corresponding to the branches in the selection. The dual problem occurs when a variable is defined in multiple branches within a selection but is used outside of it, such as in the merge process:

$$*[C?c; [c \longrightarrow Y?x [] \neg c \longrightarrow Z?x]; X!x]$$

Here, we introduce a ϕ-function to correctly merge the values from the branches so that the downstream program can function correctly:

$$\begin{aligned}&*[C?c; [c \longrightarrow Y?x_1 [] \neg c \longrightarrow Z?x_2];\\ &\quad x := \phi(x_1, x_2); X!x]\end{aligned}$$

Finally, we also introduce loop-ϕ-functions to handle loop-carried dependencies correctly. Consider a loop that executes several times. A variable used within a loop may refer to the value from before the loop started (on the first iteration) or to the final value at the end of the loop execution (on every subsequent iteration). Consider the simple process below:

$$s := 0; *[X?x; s := s + x; Y!s \ \leftarrow \ s < 10]; S!s$$

Here, the s being referred to in the RHS of the accumulation step could be either the initial condition or the loop-carried value, depending on the iteration number. Further, the final value of s at the tail of the loop needs to be mapped to either the $S!s$ action or back around to the start of the loop depending on whether the loop terminates. This is reminiscent of a ϕ^{-1}-function. In essence, a loop-ϕ-function is a ϕ and a ϕ^{-1}-function, grouped together for convenience:

$$\begin{aligned}&s_{pre} := 0;\\ &*[s_{in} := \phi_L(s_{pre}, s_{loop}); X?x; s_{out} := s_{in} + x; Y!s_{out};\\ &\quad \{s_{loop}, s_{post}\} := \phi_L^{-1}(s_{out}) \leftarrow \ s_{out} < 10]; S!s_{post}\end{aligned}$$

The variable s_{loop} is a temporary variable that is used to loop the value at the end of the loop back to the start in cases where the loop re-executes.

These special functions have been introduced to enable analysis only, and are not true executable CHP constructs in the same way that assignments, sends or receives are; they do not have a circuit implementation. Further, we only consider do-loops, and do not allow nesting of loops. This is not a true restriction as any CHP program can be systematically rewritten to satisfy this constraint (detailed in Appendix II). STF and other related canonical forms derive from dataflow languages, which describe computation in terms of functional blocks that operate on elementary values. Translating sequential programs into dataflow primitives naturally allows the concurrent execution of non-interfering parts [9]. For concreteness, we use the ϕ/ϕ^{-1} notation in STF in what follows.

III. Proposed Decomposition Technique

The first step in the automatic decomposition process is to parse the input CHP program into an abstract syntax tree (AST), which is an in-memory representation of the program. Next, we apply standard compiler optimization passes to the the AST, such as dead-code elimination and constant propagation, which have been extensively studied in the software compiler literature [10]. Following this, we perform a rewrite of every selection construct in the CHP such that the guards are in a standard form. This standard form constitutes computing and assigning n boolean variables, one for each branch of an n-way selection. As an illustration, the following CHP:

$$*[\ldots\ [x = 0 \longrightarrow ..\talloblong x = 1 \wedge y = 0 \longrightarrow ..];\ \ldots]$$

would get rewritten into:

$$*[\ldots\ (g_0 := (x = 0), g_1 := (x = 1 \wedge y = 0)); \\ [g_0 \longrightarrow ..\talloblong g_1 \longrightarrow ..];\ \ldots]$$

It should be clear that this transformation is correct when the guards consist purely of local variables, since the newly introduced variables are not visible to the rest of the system. Note that since we are focused on decomposing programs that are slack elastic, the assumption that guards are probe-free is a natural one; the presence of probes in guards can be a source of non-determinism and slack in-elastic CHP programs [4]. We apply this particular transformation because it exposes opportunities for decomposition by separating the computation of the guard from the actual process of choosing a branch in the selection itself. Following this, we convert the AST into static token form, which forms the foundation for the data-dependency analysis.

A. Graphs and Data Dependencies

Once the AST has been placed into STF, we construct an auxiliary data structure, the dependence-based decomposition graph (DDG), which is the entity that actually forms the basis of the decomposition. The vertices/nodes in this graph are one of the following:

- Basic node, corresponding to a basic action in the CHP—send, receive, or assignment.
- Guard node, which is a 'dummy' node that is used to encode a particular branch of a selection. Including this enables additional decomposition opportunities, as we will show later, and is a new feature of our analysis compared to more traditional control- and data-dependency graphs.
- Selection ϕ-node, corresponding to a ϕ-function.
- Selection ϕ^{-1}-node, corresponding to a ϕ^{-1}-function.
- Loop ϕ-node, corresponding to a ϕ-function for loops.

TABLE I
Variables that are used and defined by different node types

Node Type	CHP	Defs	Uses
Receive	$C?x$	x	-
Assign	$x := e$	x	vars. in e
Send	$C!e$	-	vars. in e
Sel. ϕ	$x := \phi(x_1, .., x_n)$	x	$x_1, ..x_n$
Sel. ϕ^{-1}	$\{x_1, .., x_n\} := \phi^{-1}(x)$	$x_1, ..x_n$	x
Loop ϕ	$x_{in} := \phi_L(x_{pre}, x_{loop})$	x_{in}	x_{pre}, x_{loop}
Loop ϕ^{-1}	$\{x_{loop}, x_{post}\} := \phi_L^{-1}(x_{out})$	x_{loop}, x_{post}	x_{out}
Guard	g (guard expr.)	-	vars. in g

- Loop ϕ^{-1}-node, corresponding to a ϕ^{-1}-function for loops.

Each node in the graph is associated with two sets of variables - the set of variables it defines (`defs`) and the set of variables it uses (`uses`). The `defs` is the set of variables that are assigned/written to, in that node. For example, an assignment node $z := f(x, y)$ has the `defs` set $\{z\}$. The `uses` is the set of variables whose values are read/used by that node. In the previous example, the `uses` set is $\{x, y\}$ as values of both of these are used to compute the RHS of the assignment. Note that one or both of these sets might be empty. Table I specifies exactly the `defs` and `uses` set for every type of node in the graph.

The edges between nodes in this graph are directed, and represent data-dependencies. An edge e from node n_1 to node n_2 exists if one of the three following conditions are met:

1) When n_1 defines a variable that n_2 then uses.
2) If n_1 is a guard node and n_2 is a ϕ^{-1}-node of the same selection, and if the output of the ϕ^{-1} corresponding to the branch of n_1 is non-null.
3) If n_1 is a guard node and n_2 is a ϕ-node of the same selection, and if the input of the ϕ corresponding to the branch of n_1 is defined within the selection.
4) If n_1 is a guard node of a loop and n_2 is a ϕ^{-1}-node or a ϕ-node of the same loop.

This DDG captures all of the information about the data dependencies between different blocks in the program, and allows analysis without the syntactic restrictions of the sequential program. Next, we compute the graph of strongly-connected components (SCC-graph) of the DDG. A strongly-connected component of a directed graph is a maximal subset of the vertices such that there is a directed path from every vertex in the subset to every other vertex. The SCC-graph is a graph such that each node in it represents an SCC in the original DDG. The next section describes how strongly-

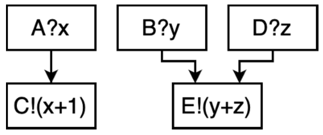

Fig. 1. Dependence-based Decomposition Graph for the example CHP program. This is a DAG and hence is its own SCC-graph.

connected components in the DDG capture generalized loops and why we use this approach. Note that in a directed acyclic graph (DAG), which corresponds to programs with no loop-carried variables, every node is trivially its own SCC and the DAG is its own SCC-graph. Hence, we will use the terms interchangeably for DAGs. Following this, we compute the weakly-connected components of the SCC-graph. A weakly-connected component of a directed graph is a subset of vertices, such that when considering the directed graph as an undirected graph, every vertex within the subset can be reached from every other within the subset, but cannot be reached from any vertex outside the subset. These components capture independent data-dependence domains in the program. At this point in the flow, there are two possibilities. Firstly, the entire directed graph is comprised of a single weakly-connected component. In this case, which will be discussed later, the process cannot be decomposed into several sub-processes free of cost. The other case is when there is more than one weakly-connected component. Here, each component can be decomposed into a separate process since the lack of edges between the components implies that there are no data dependencies between any pair. For example, consider the following simple CHP:

$$*[A?x; B?y; C!(x+1), D?z; E!(y+z)]$$

The DDG for this program is shown in Fig. 1. All nodes are basic nodes and the edges are added according to the aforementioned rules. For example, an edge from $B?y$ to $E!(y+z)$ exists since the latter uses a variable that the former defines. The DDG for this program consists of two components. Here, the original program can be rewritten into two concurrent programs, as follows:

$$*[B?y; D?z; E!(y+z)] \quad \| \quad *[A?x; C!(x+1)]$$

However, as mentioned earlier, it is not necessary that the program neatly separates into several components. With large sequential programs, the data dependencies are typically intricate and weave through the length of the program and some transformations of the DDG need to be performed in order to enable decomposition.

Intuitively, it should be clear that each weakly-connected component of the DDG can be implemented as an individual process. Since there are no shared variables between the components (as there are no edges between them), the sub-processes can be viewed as a projection of the original process onto disjoint sets of variables. Hence we know that this collection of sub-processes is equivalent to the original CHP by the projection theorem [5].

B. Transformations

The previous section covered the case where the process decomposition could be performed at no cost. While this is beneficial, it is often the case that introducing some additional communications could result in a process that can be decomposed into simpler sub-processes. As an example, consider the following CHP:

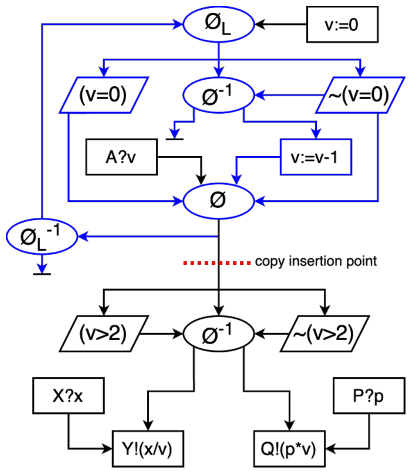

Fig. 2. Dependence-based Decomposition Graph for the example CHP program with loop-carried dependencies. Null-terminated edges do not actually exist in the graph as they do not define a variable, and are shown only for clarity. Nodes in blue are part of the same SCC and hence the blue edges, which are within the same SCC are ignored. The nodes in black each form their own SCC. Since the loop guard is simply $true$ in this case, we omit the loop-guard node and its associated edges from the diagram.

```
$v := 0;$
$*[[v = 0 \longrightarrow A?v [] else \longrightarrow v := v - 1];$
  $[v > 2 \longrightarrow X?x; Y!(x/v)$
  $[] else \longrightarrow P?p; Q!(p*v)]$ ]
```

Notice that the variable v is a loop-carried dependency for the entire process but is not actually modified in the second selection at all. If this process were used in an environment that can produce/consume tokens sufficiently quickly on all input/output channels, then the local cycle time of this process would limit throughput since the execution of the first selection statement in iteration $(i+1)$ of the loop would have to wait for the execution of the second selection statement from iteration i to complete.

The DDG of this process is shown in Fig. 2. Here, the nodes in blue form an SCC, and hence the SCC-graph of this DDG simply has this set of blue nodes replaced by a single node. The nodes in black form their own atomic SCC. From the DDG and the corresponding CHP, notice that the final write to the variable v is in the first selection and the rest of the program only reads it. Due to this fact, the portion of the process that updates v, i.e. the first selection, can actually proceed to compute the next value while the other portion, i.e. the second selection, is still performing its computation using the first value of v, provided there exists a copy of v for the second selection to use. This is resolved by inserting a copy, i.e. breaking the edge from the ϕ-node for v and introducing two nodes, which send and receive the necessary value. In the CHP, this is exactly the same as introducing an internal communication (copy) and corresponding renaming of the variable downstream in the program, relative to the point of the communication.

Another way to see this in terms of the DDG. The copy insertion point shown in Fig. 2 partitions the DDG into two components. Furthermore, all edges from the original DDG are from one component to the other—i.e. the graph of components is acyclic. This means that the first component can run ahead of the second component, as long as there is slack on the edge that contains the copy-insertion point. When performing a copy-insertion in the DDG, we also perform the corresponding change in the AST for convenience later in the flow. Consequently, the copy-insertion process modifies the CHP, resulting in:

$$\begin{array}{l} v := 0; \\ *[[v = 0 \longrightarrow A?v \,[]\, \mathit{else} \longrightarrow v := v - 1]; \\ \quad (V!v,\, V?v_1); \\ \quad [v_1 > 2 \longrightarrow X?x;\, Y!(x/v_1) \\ \quad [] \,\mathit{else} \longrightarrow P?p;\, Q!(p*v_1)] \;] \end{array}$$

At this point, the DDG has two weakly connected components, resulting from the copy insertion, and can be decomposed into two concurrent programs, which each have fewer actions in series, resulting in a smaller cycle time, at the cost of added communication actions:

$$\begin{array}{l} v := 0; *[[v = 0 \longrightarrow A?v \,[]\, \mathit{else} \longrightarrow v := v - 1];\, V!v] \\ \| *[V?v_1; \\ \quad\quad [v_1 > 2 \longrightarrow X?x;\, Y!(x/v_1) \\ \quad\quad [] \,\mathit{else} \longrightarrow P?p;\, Q!(p*v_1)] \;] \end{array}$$

The second process still depends on the *value* of v that is computed by the first process. However, instead of the data-dependence being captured by edges in the DDG, i.e. local variables, it is captured by a send-receive pair, i.e. a channel between processes. This process of inserting copies and moving the data-dependence to the channel level opens up the possibility for creating smaller processes from the original program as well as adding slack on the newly introduced channel to decouple the two processes.

The reason we compute the SCC-graph is to forbid copy-insertion on edges *within* the same SCC. Inserting a copy within an SCC, which captures the loop-carried nature of v, will exacerbate this loop latency due to the overhead of communication actions, while likely providing no throughput benefit. Since we expect the synthesized circuits to be latency-limited, this is a reasonable constraint to include. Further, this grouping allows us to treat several dependencies across the same two SCCs at once, as sets of edges between two SCCs would be condensed into a single edge in the SCC-graph.

The methods described thus far have only dealt with selections as an entire atomic unit. However, in some cases, it may be advantageous to decompose different branches of a selection into different processes. The DDG elegantly allows this as well through the use of the guard nodes. Guard nodes decouple the computation of the guard from the actual choice of branch. In order to illustrate this, consider the following example:

$$\begin{array}{l} *[A?x, B?y, D?z, C?c; \\ \quad [c = 0 \longrightarrow X!x \quad\quad [] \, c = 1 \longrightarrow Y!y \\ \quad [] \, c = 2 \longrightarrow Z!(y+z) \,[]\, c = 3 \longrightarrow \mathit{skip}] \;] \end{array}$$

Here, the four branches of the selection in the program consist of two disjoint sets of variables, $\{x\}$ and $\{y, z\}$.

This allows for the selection to broken into two separate selections in two processes. The crucial variable, as can be seen from the DDG in Fig. 3, is c, which is used to compute the guards. If copy insertion is performed on the edge leading into the guard nodes, then the DDG can be split into two components. When the processes are reconstructed, the resulting CHP is:

$$\begin{array}{l} *[A?x, C_1?c_1; \\ \quad [c_1 = 0 \longrightarrow X!x \;\, [] \, \mathit{else} \longrightarrow \mathit{skip}] \;] \quad \| \\ *[B?y, D?z, C?c;\, C_1!c; \\ \quad [c = 1 \longrightarrow Y!y \,[]\, c = 2 \longrightarrow Z!(y+z) \\ \quad [] \, c = 3 \longrightarrow \mathit{skip} \,[]\, \mathit{else} \longrightarrow \mathit{skip}] \;] \end{array}$$

When a selection is fractured into two or more selections in this manner, it is also necessary to introduce an else-skip branch in every new selection, if it does not already have one.

Algorithm 1 Decomposition Algorithm

```
1: Transform selections in CHP into standard form.
2: Place CHP into static token form.
3: Build DDG $D$ from CHP.
4: Compute SCC-graph $D_S$ of $D$.
5: $W \leftarrow WCC(D_S)$          ▷ Weakly-connected components
6: for $w \in W$ do
7:     for $e$ $(n_1 \to n_2) \in w$ do                ▷ $e$ : edge
8:         if $\neg(n_1$=receive$|n_2$=send$) \wedge$ F$(w, e, n_1, n_2)$ then
9:             go to 16
10:        end if
11:    end for
12:    for $e$ $(n_1 \to n_2) \in w$ do
13:        if $(n_1$=receive$|n_2$=send$) \wedge$ F$(w, e, n_1, n_2)$ then
14:            break
15:        end if
16:    end for
17: end for
18: Rebuild CHP from graph D
19:
20: function F$(w, e, n_1, n_2)$
21:    $w' \leftarrow w \setminus \{e\}$          ▷ $w$ with edge $e$ removed
22:    if $|WCC(w')| > 1$ then
23:        delete $(e)$
24:        insert_copy $(n_1, n_2)$
25:        return true
26:    end if
27:    return false
28: end function
```

The algorithm used to determine the edges to perform copy-insertion on is shown in Algorithm 1. The function $WCC(\cdot)$ returns the weakly-connected components of a graph, and is implemented using a simple union-find data structure.

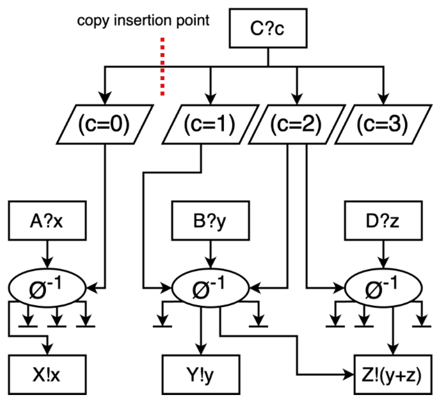

Fig. 3. Dependence-based Decomposition Graph for the example CHP program with disjoint sets of variables in selection branches. Null-terminated edges do not actually exist in the graph as they do not define a variable, and are shown only for clarity. This is a DAG and hence is its own SCC-graph.

Following this, there are two passes. The core decision, in line 22, is the same for both passes, and checks whether deleting that particular edge would increase the number of weakly-connected components in the graph. If so, the copy-insertion is performed. The difference between the passes is that on the first pass, edges with receives as source nodes or sends as destination nodes are ignored, since performing copy-insertion on these edges would be equivalent to adding a buffer on that channel, and better candidates for decomposition might be ignored. If no candidates were found in the first pass, then the excluded edges are iterated over in the second pass to potentially find a candidate.

Notice that the algorithm is guaranteed to perform at least one copy-insertion, given that the original process has at least one channel connected to it. This condition is satisfied by every process of interest since otherwise, the process would be unobservable by the rest of the processes in the system and thus can be removed entirely without affecting the functionality of the system itself. Finally, the algorithm can be run iteratively any number of times (chosen by the user) on the CHP to obtain further pipelined/decomposed CHP. Once the final DDG after copy-insertion is obtained, the CHP for each process is constructed from it in the following way. For each weakly-connected component w, we iterate through a copy of the AST for the CHP and delete every node that is not present in w. As mentioned earlier, the send and receive nodes corresponding to copy-insertions are already present in the AST, and thus no extra work needs to be done to insert those in the appropriate locations. The resulting pruned AST is then printed out as a single CHP process.

C. Relation to Dataflow Decomposition

As mentioned earlier in the section, it is rarely the case that input sequential program intrinsically contains disjoint subgraphs that can be extracted into their own processes. More often, copy-insertion is necessary to decompose the original program into parts. Hence, the power of the method originates from identifying suitable points to perform this operation. In our current implementation, the choice of copy-insertion location is made according to a simple heuristic: if deleting an edge and placing a copy increases the number of connected components, then that communication pair is inserted. This step forms the crucial iterative process in adding concurrency to the program. Notably, if this is performed repeatedly on an acyclic DDG by inserting copies at every edge, the system reaches a fixed point where each process becomes extremely simple. At this point, every process will be one of the dataflow components, and further decomposition will only result in adding buffers on the channels that exist in this system. In the event the original DDG has cycles, we can break all loops with the introduction of initial token buffers (see [3, 6]); this transformation will result in a DDG that is always acyclic. Hence, the iterative decomposition we describe here can be thought of as being a stepwise path to fine-grained dataflow pipelining, where we have the opportunity to stop at any point along the path—from fully sequential to fully pipelined.

IV. Results

A software tool has been developed that automatically performs the aforementioned processing on the CHP description of a given set of processes, and produces as output a more concurrent CHP description of the same. The tool leverages the ACT open-source EDA flow for asynchronous circuits [11]. ACT provides native support for CHP and the generated gate-level descriptions in terms of production rules. In addition, ACT also has tools that can export designs into verilog netlists as well as LEF/DEF for compatibility with commercial place-and-route flows. The circuit description can also be exported to SPICE for simulation. This allows a designer to use a combination of ACT and commercial tools for design, testing and implementation of ASICs.

In order to benchmark the original (sequential) description against the output (concurrent) description, we synthesize both using Maelstrom [12], which has been shown to have a significant improvement over Balsa [2], and thus provides a good estimate of performance numbers from CHP descriptions. In particular, when presented with CHP programs that correspond to dataflow components, Maelstrom produces circuits that have roughly the same energy, delay, and area as those corresponding to hand-optimized dataflow components [12].

Maelstrom is a state-of-the-art synthesis technique for asynchronous circuits that produces highly efficient control circuits to implement a high-level CHP program. The datapath implementation supports several types, such as latches, edge-triggered flip-flops and QDI registers. To synthesize expressions into combinational circuits, Maelstrom exports the expression into Verilog syntax and uses external logic synthesis tools. In particular, both open-source tools such as `abc` [13] and commercial tools like Cadence's `genus` are supported, thereby leveraging decades of investment in this area from the synchronous EDA community. The results from the external logic synthesis are then imported to produce the complete circuit.

We synthesize simple CHP programs in order to illustrate some key advantages of concurrent decomposition, as well as larger designs to demonstrate quantifiable performance improvements.

Figure 4 summarizes the results from performing decomposition on the CHP followed by circuit synthesis. We use a 65nm technology for all benchmarks. Area estimates are from a placed, power-routed and global-routed design. Delay and power estimates are calculated from SPICE simulations of the synthesized circuit. The rows denoted Decomposed 1, 2 and 3 refer to running the iterative decomposition procedure once, twice and thrice respectively. This choice was due to the fact that the gains plateau after 3 iterations for the processes tested. Note that we provide power estimates in place of energy per cycle due to the fact that in some cases, there are several data-independent parts of the program that decompose out into circuits that run in parallel without any synchronization between them, and energy per output token is ill-defined. In these cases, the throughput of the entire system is also ill-defined, since there are now more than one. As a result, for these cases, we report throughput numbers of every independent component.

From Fig. 4, notice that for complex programs, decomposition typically results in higher throughput at the cost of a larger circuit, increased latency and power consumption. The source of the added circuitry is the added communications that are inserted while performing copy-insertions. Further, note that the gains in throughput plateaus at different points for different programs. This is due to the fact that beyond a point, the only available points to insert copies at are sends or receives. As mentioned earlier, this simply increments the slack on that channel (adds a buffer), and thus increases the latency without providing a significant throughput benefit, as no logic pipelining is achieved.

An interesting observation is that in some cases, such as the first two example CHP programs, the area of the synthesized circuit actually reduces after one level of decomposition, due to the fact that several unnecessary sequencings that existed in the original program do not exist anymore. This results in simpler wiring that enables a tighter placement, thereby bringing down the overall area.

V. Future Work

As mentioned at the end of Section III, the copy insertion strategy is currently based on a simple heuristic. As a result, this might not always result in the best possible decomposition for a given input. In some cases, it might be detrimental to perform any decomposition at all and this is not an option in the heuristic as it always performs at least one copy-insertion. Even if decomposition is necessary, it is difficult to determine the optimal extent as this is dependent on the input, as can be seen from Table 4.

Furthermore, the heuristic does not have any information about the cost of performing a copy-insertion at a point built into it, as can be seen from Algorithm 1. If there are multiple edges that can be deleted to be to convert a single weakly-connected component into two, then these are treated as equal by the algorithm. It may be the case that some deletions result in superior circuits as compared to others, and this needs to be taken into account. The problem is also exacerbated by the fact that the purpose of decomposition is typically to hit some target metric (throughput, energy, cycle time etc.) and this is not a considered factor when performing the iterative process. We are currently investigating improved decomposition heuristics that are driven by user requirements (e.g. an area budget).

Finally, our current work focuses on improving the throughput of the system only. Orthogonal to this, there has been work on sharing expensive hardware resources without sacrificing performance [15, 16]. Incorporating resource sharing and scheduling of operators into the framework that we have developed here forms another important direction for future research, and would enable further improvement in the quality of synthesized circuits.

VI. Conclusion

In this work, we presented a novel, completely automated method to decompose a sequential specification of an asynchronous circuit into a concurrent one, with control over the degree of concurrency. The decomposition technique allows the circuit designer to target a particular degree of concurrency, instead of being restricted to either extreme—sequential or dataflow synthesis. Our technique provides a concurrent but equivalent description of the input program, thereby allowing the generated circuits to have higher throughput, at the cost of higher area and power consumption. The separation of decomposition and synthesis into distinct steps also allows for a clearer understanding of the behavior of the system at the abstract and circuit level. We demonstrate that our decomposition procedure, when applied to an input CHP specification, results in superior circuits than those that would be possible with a direct sequential synthesis. The decomposition relies on a heuristic to split a program into concurrent parts, which can be significantly improved by using post-synthesis circuit information in a feedback loop to inform its decisions in exploring the search-space.

Appendix I

Communicating Hardware Processes (CHP) is a hardware description language used to describe clockless circuits that is derived from Hoare's Communicating Sequential Processes (CSP) [17]. A full description of CHP and its semantics can be found in [18]. Below is an informal description of a subset of that notation that we use, listed in descending precedence, replicated from [19]. For a complete discussion of the interaction between the handshake expansions of channel actions and the composition operators, see [20].

A **Channel** X consists of a **request** X.r and either an **acknowledge** X.a or **enable** X.e. The acknowledge and enable serve the same purpose, but have inverted sense. With these

Test Program	CHP Description	Synthesis Method	Area (μm^2)	Latency (ns)	Throughput (MHz)	Power (μW)
Sequential Buffers	$*[A?w;\ W!w;$	Sequential	3833	2.84	219	**112**
	$B?x;\ X!x;$	Decomposed 1	**3040**	**0.18** (x4)	**2810** (x4)	405
	$C?y;\ Y!y;$	Decomposed 2	6067	0.35 (x4)	2700 (x4)	793
	$D?z;\ Z!z]$	Decomposed 3	12125	0.74 (x4)	2670 (x4)	1520
Linear Subprograms	$*[A?x;$	Sequential	3258	1.26	271	**138**
	$B?y;$	Decomposed 1	**3071**	**0.75, 1.35**	559, 400	304
	$C!(x+1), D?z;$	Decomposed 2	4897	0.94, 2.11	**563, 408**	442
	$E!(y+z)]$	Decomposed 3	7807	1.23, 2.84	559, 403	538
Conditional Router	$*[C?c, A?x, B?y;$	Sequential	**2954**	**1.12**	195	**153**
	$[c=0 \to X!x$	Decomposed 1	3849	2.04	437	399
	$[]\, c=1 \to Y!y]]$	Decomposed 2	5483	3.41	**444**	629
		Decomposed 3	8282	5.13	442	903
MIPS R3000 Writeback Unit	CHP exactly as in [5]	Sequential	**7081**	**1.12**	282	**280**
		Decomposed 1	9090	4.32	331	398
		Decomposed 2	12392	5.70	391	613
		Decomposed 3	18093	7.49	**392**	932
Async. RISC-V Fetch Unit	Microarchitecture detailed in [14]	Sequential	**44731**	**2.77**	141	**875**
		Decomposed 1	58897	4.52	248	1022
		Decomposed 2	84826	7.01	251	1321
		Decomposed 3	105899	8.85	**254**	1873

Fig. 4. Synthesis results for CHP programs using Maelstrom, without and with varying levels of the automated decomposition. All metrics are from SPICE simulations in a 65nm technology node. Note that for the first test-case, the resulting decomposed processes are 4 identical and unsynchronized copies of the same template, i.e. a buffer, and hence we report it once, with an (x4) marker. For the second test-case, the two resulting processes are entirely unsynchronized relative to each other, and hence we report both individual throughput and latency values.

signals, a channel implements a network protocol to transmit data from one process to another.

- **Skip**: *skip* does nothing; continues to the next command.
- **Assignment**: $x := e$ sets the variable x to the value e, where e is an expression.
- **Send**: $C!e$ sends the value of e over the channel C.
- **Receive**: $C?x$ receives a value over channel C and stores it in the variable x.
- **Sequential Composition**: $S; T$ executes the programs S followed by T.
- **Parallel Composition**: $S \,||\, T$ executes the programs S and T in any order.
- **Deterministic Selection**: $[G_1 \to S_1 [] ... [] G_n \to S_n]$ where G_i, called a guard, is a dataless expression and S_i is a program. The selection waits until one of the guards, G_i, evaluates to $true$, then executes the corresponding program, S_i. The guards must be stable and mutually exclusive. The notation $[G]$ is shorthand for $[G \to skip]$, which corresponds to waiting for G to become true.
- **Non-Deterministic Selection**: $[|\, G_1 \to S_1 [] ... [] G_n \to S_n \,|]$ is the same as Deterministic Selection except that the guards do not have to be stable or mutually exclusive. If two or more evaluate to $true$ simultaneously, then one is picked arbitrarily (not necessarily random).
- **Loop**: $*[G_1 \to S_1 [] ... [] G_n \to S_n]$ is similar to the selection statements. However, once a branch is completed, the guards are re-evaluated and another branch is chosen. The process is repeated until no guard evaluates to $true$, in which case the loop terminates. $*[S]$ is shorthand for $[true \to S]$.
- **Do-Loop**: $*[S_1 \leftarrow G_1]$ is similar to the loop, but can have only one branch. The branch is executed before evaluating the guard, similar to a do-while loop in software programming languages.

Appendix II

Loops that are nested within other loops can be systematically extracted. Consider a loop within another as follows:

$$*[..; *[G_1 \longrightarrow S_1 [] G_2 \longrightarrow S_2 .. [] G_n \longrightarrow S_n]; ..]$$

This CHP is rewritten into two concurrent processes as:

$$\begin{aligned}
&*[..; L_s!\{x_1, x_2, .., x_n\}, L_f?\{y_1, y_2, .., y_m\}; ..] \\
&\| \quad c := 0; \\
&\quad *[\ \ [c = 0 \longrightarrow L_s?\{x_1', x_2', .., x_n'\}, c := 1 \\
&\qquad\ \ [] c = 1 \longrightarrow skip]; \\
&\qquad [G_1' \longrightarrow S_1' [] G_2' \longrightarrow S_2' .. [] G_n' \longrightarrow S_n' \\
&\qquad\ \ [] else \longrightarrow L_f!\{y_1', y_2', .., y_m'\}, c := 0]\ \ ]
\end{aligned}$$

where G_i' and S_i' are the same as G_i and S_i respectively, with the variables replaced by their appropriate primed counterparts. The sets of variables $\{x_i\}$ and $\{y_i\}$ are those are needed within the loop computation (live-in to the loop) and produced by the loop computation (live-out of the loop) respectively. Next, we consider the transformation of multiple-branch loops into do-loops. Suppose there exists a loop:

$$*[G_1 \longrightarrow S_1 [] G_2 \longrightarrow S_2 .. [] G_n \longrightarrow S_n]$$

This can be rewritten into a do-loop, by embedding a selection within it:

$$\begin{aligned}
c := 1; *[[G_1 \longrightarrow S_1 [] G_2 \longrightarrow S_2 .. [] G_n \longrightarrow S_n \\
[] else \longrightarrow c := 0] \leftarrow (c = 1)]
\end{aligned}$$

References

[1] R. Manohar, *Chp2prs: Syntax-directed translation of CHP programs into production rules*, 2023.

[2] A. Bardsley and D. Edwards, “The balsa asynchronous circuit synthesis system,” in *Forum on Design Languages*, 2000.

[3] R. Li, L. Berkley, Y. Yang, and R. Manohar, “Fluid: An Asynchronous High-level Synthesis Tool for Complex Program Structures,” in *2021 27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, Beijing, China: IEEE, 2021.

[4] R. Manohar and A. J. Martin, “Slack elasticity in concurrent computing,” in *Mathematics of Program Construction*, G. Goos, J. Hartmanis, J. Van Leeuwen, and J. Jeuring, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[5] R. Manohar, T.-K. Lee, and A. J. Martin, “Projection: A synthesis technique for concurrent systems,” in *Proceedings. Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Barcelona, Spain: IEEE Comput. Soc, 1999.

[6] J. Teifel and R. Manohar, “Static tokens: Using dataflow to automate concurrent pipeline synthesis,” in *10th International Symposium on Asynchronous Circuits and Systems, 2004. Proceedings.*, Crete, Greece: IEEE, 2004.

[7] C. G. Wong and A. J. Martin, “High-level synthesis of asynchronous systems by data-driven decomposition,” in *Proceedings of the 40th annual Design Automation Conference*, Anaheim CA USA: ACM, 2003.

[8] C. S. Ananian, “The static single information form,” Ph.D. dissertation, Massachusetts Institute of Technology, 2001.

[9] J. B. Dennis, “First version of a data flow procedure language,” in *Programming Symposium*, B. Robinet, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1974.

[10] A. V. Aho, M. S. Lam, and J. D. Ullman, *Compilers: Principles, Techniques & Tools*. London, England: Pearson Education, 2007.

[11] S. Ataei *et al.*, “An Open-Source EDA Flow for Asynchronous Logic,” *IEEE Design & Test*, 2021.

[12] K. Srinivasan and R. Manohar, “Maelstrom: A logic synthesis technique for asynchronous circuits,” in *International Workshop on Logic Synthesis (IWLS) (poster)*, 2024.

[13] R. Brayton and A. Mishchenko, “ABC: An Academic Industrial-Strength Verification Tool,” in *Computer Aided Verification*, D. Hutchison *et al.*, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[14] R. Dashkin, “Asynchronous risc-v cpu design with pre-silicon validation on synchronous fpgas,” Ph.D. dissertation, Yale University, 2024.

[15] J. Hansen and M. Singh, “A fast branch-and-bound approach to high-level synthesis of asynchronous systems,” in *2010 IEEE Symposium on Asynchronous Circuits and Systems*, 2010.

[16] J. Hansen and M. Singh, “A fast hierarchical approach to resource sharing in pipelined asynchronous systems,” in *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*, 2012.

[17] C. A. R. Hoare, “Communicating sequential processes,” *Communications of the ACM*, 1978.

[18] A. J. Martin, “Synthesis of asynchronous VLSI circuits,” California Institute of Technology, Tech. Rep., 1991.

[19] N. Bingham and R. Manohar, “A Systematic Approach for Arbitration Expressions,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2020.

[20] R. Manohar, “An analysis of reshuffled handshaking expansions,” in *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001*, Salt Lake City, UT, USA: IEEE Comput. Soc, 2001.