Static Tokens: Using Dataflow to Automate Concurrent Pipeline Synthesis

John Teifel and Rajit Manohar
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853, U.S.A.

Abstract

We describe a new intermediate compiler representation, static token form, that is suitable for dataflow-style synthesis of high-level asynchronous specifications. Static token form transforms variables into tokens, and simplifies the pipelining of asynchronous computations according to their dataflow graphs. We present a compiler framework that automates this synthesis method, and show how it can be applied to pipelined asynchronous FPGA architectures.

1. Introduction

The formal synthesis method of asynchronous design transforms a high-level sequential hardware specification into a system of concurrent processes. How the sequential specification is partitioned into these processes depends on the particular design point (i.e., area, energy, latency, throughput, etc.) desired for the resulting system. This decomposition procedure is difficult to do optimally for more than one relevant design parameter and in most full-custom asynchronous designs (e.g., [12]) it is largely done manually to maintain tight control over performance. In asynchronous systems that do not need full-custom performance, however, automated process decomposition tools that reduce design time are more important than a hand-optimized process decomposition. In this paper, we describe a hardware compiler that uses sequential compiler analyses to automatically decompose a sequential hardware description into highly concurrent pipelined circuits.

Recent advances in the design of pipelined asynchronous FPGA architectures [15, 16] motivated the development of this hardware compiler. These asynchronous FPGAs have simple fine-grain programmable pipeline stages that can be configured to compute functions, copy tokens, or to merge and split token streams. To use these FPGAs efficiently, we needed a new decomposition method that produced simple fine-grain processes from high-level logic specifications.

Previous syntax-directed synthesis methods (e.g., [3, 5, 18]) fail to produce such pipelines because they cannot take advantage of fine-grain concurrency that is hidden in high-level sequential program descriptions. Similarly, the *data-driven decomposition* method [19], which projects each high-level variable onto its own process, does not produce processes that are simple enough to be directly implemented on a pipelined asynchronous FPGA architecture.

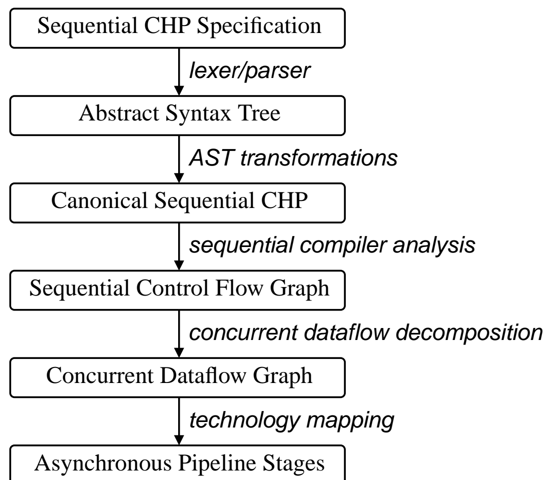

Figure 1. Concurrent pipeline synthesis steps.

In contrast, our compiler generates concurrent processes using only seven types of simple pipeline templates, making it suitable for both FPGA and custom logic synthesis. Our main insight was to treat variable definitions as producers of data tokens and uses of variables as consumers of data tokens. The compiler, however, must *statically* guarantee that token producers and token consumers will cooperate to correctly implement the original sequential program.

Figure 1 shows the pipeline synthesis steps automated by our compiler. The front-end of the compiler accepts high-level sequential logic specifications written in the CHP hardware description language, whose syntax is described in Appendix A. Section 2 describes the Abstract Syntax Tree (AST) transformations that are necessary to convert

$$
\begin{array}{l}
\ldots \\
[\; G_0 \longrightarrow \ldots \\
\| \; G_1 \longrightarrow \ldots \\
\| \; \ldots \\
\| \; G_{n-1} \longrightarrow \ldots \\
] \\
\ldots
\end{array}
\qquad
\begin{array}{l}
\ldots \\
g := selection(\ldots); \\
[\; g = 0 \longrightarrow \ldots \\
\| \; g = 1 \longrightarrow \ldots \\
\| \; \ldots \\
\| \; g = n-1 \longrightarrow \ldots \\
] \\
\ldots
\end{array}
$$

Figure 2. Canonical CHP form: original selection (left) and selection with integer guard expressions (right).

a parsed CHP program into the canonical CHP form. In Section 3, we introduce the Static Token (ST) intermediate compiler representation to analyze the sequential behavior of a canonical CHP program and to produce an optimized sequential control flow graph. Using concurrent dataflow decomposition, in Section 4, we transform the control flow graph into a concurrent dataflow graph. The concurrent dataflow graph defines a set of concurrent hardware processes that is functionally equivalent to the original sequential CHP specification and can be directly mapped to an asynchronous FPGA or to custom circuits.

The remaining sections of this paper are organized as follows. In Section 5, we present correctness conditions when concurrent dataflow decomposition can be safely applied. Section 6 discusses the implementation of the compiler and our experience in using it to synthesize logic for pipelined asynchronous FPGAs. We compare related work in Section 7 and conclude in Section 8. Common compiler terminology is defined in Appendix B.

2. Canonical CHP Form

To simplify the back-end steps of our pipeline compiler, we convert the input CHP specification into *canonical* CHP form. Canonical CHP form is a subset of CHP having the following restrictions: guard expressions are of the form "$v = i$," where v is an integer variable and i is a constant; all loops are either infinite repetitions (i.e., the outermost loop), or contain only a single guard of the form $v = 1$; every channel action appears at most once in the body of a CHP process. The first two requirements are a technicality, and are used to simplify the presentation.

All guard expressions in canonical CHP form must be rewritten to contain only integer values as shown in the right part of Figure 2, where $selection$ is a function that computes the guard variable g corresponding to the matching guard expression in the original CHP program. Unique channel actions imply that only a single channel action per port can appear in the main loop of a sequential program (e.g., $*[Z?x; Z!x]$ is allowed, but not $*[A?x; Z!x; Z!x]$). Syntactic AST transformations to remove multiple channel actions are described in Appendix C. Note that canonical CHP allows an additional channel send action to appear in the initialization part of the main program loop.

3. Static Token Form

The goal of the sequential analysis phase of our compiler is to transform a canonical CHP program into a form that we call the *static token* (ST) form. ST form is an extension of the Static Single Information (SSI) intermediate compiler representation [2], which in turn is a generalization of the Static Single Assignment (SSA) form [4]. The important properties of SSA and SSI form can be captured by examining definition/use (def-use) chains in a compiler. In what follows, we examine the properties of these representations by examining the control flow graph (CFG) of the CHP program.

In SSA form, each variable use in the program has a single reaching definition. If there are multiple reaching definitions (at merge points in the control-flow graph), then these are combined into a single definition using ϕ-functions. Also, each definition of a variable is given a fresh name, and uses are appropriately updated to reflect the appropriate (single) reaching definition.

SSI form was introduced to improve backward flow analysis, and to augment dataflow information using partial path information. In SSI form, if a definition at program point x reaches two uses at points y and z, then either all paths from x to y contain z or all paths from x to z contain y. Statements of the form $x_0, x_1 := \sigma(x)$ are used to create multiple copies of a single variable, allowing the transformed program to satisfy the SSI condition.

The concurrency-introducing transformation that we would like to apply to convert a sequential CHP program into a fine-grained concurrent version is *projection* [10]. To enable the application of projection, one of the standard sequential transformations applied is to replace assignment statements "$v := e$" with communication actions, i.e., $(X!e \parallel X?v)$ [7]. This allows us to think of variables and values as tokens that are received and sent on channels, converting data dependencies into channels. The earliest point in the program where the communication action $X!e$ can be placed is the point at which the variables used in e are defined. Therefore, one can think of definition points as generating data tokens that are transformed by expression-computation blocks, and finally consumed by the uses of the expression. The goal of static token form is to enable this interpretation of variable definitions and uses.

Static token form keeps the ϕ and σ functions of SSI form, except it re-interprets their execution as well as makes them truly executable. The statement $x := \phi(x_0, \ldots, x_{n-1})$ is now replaced with $x := \phi_g(x_0, \ldots, x_{n-1})$ where g is

an integer-valued variable. Execution of this statement is equivalent to $x := x_g$, thereby making the condition under which each argument of ϕ is used explicit. ϕ_g functions are similar to *guarded ϕ functions* [1]. The statement $x_0, \ldots, x_{n-1} := \sigma(x)$ is replaced with $x_0, \ldots, x_{n-1} := \phi_g^{-1}(x)$ where g is an integer-valued variable. Execution of this statement is equivalent to $x_g := x$, thereby making this a conditional definition. Both ϕ and ϕ^{-1} unconditionally use the variable g.

ST form introduces two new statements, $\mathbf{s}(x)$ and $\mathbf{I}_k(x)$. Statement $\mathbf{s}(x)$ uses x without defining any new variable. This statement is used to consume any unused definitions. $\mathbf{I}_k(x)$ is a statement that asserts that the initial value of x is the constant k. This statement is required to initialize variables at reset and to initialize guard variables in repetition statements.

Since ST form is generated from a CHP program, we are guaranteed that the resulting control flow graph is reducible, which simplifies the placement of the four constructs introduced by ST form. Given the restricted nature of canonical CHP, the only non-trivial control flow graph configurations possible are shown in Figure 3, where the dashed boxes correspond to control flow split and merge points.

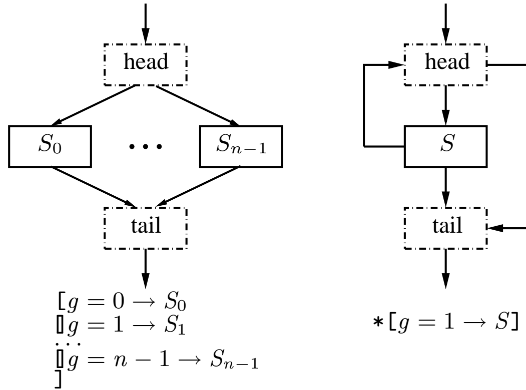

Figure 3. Possible CFG Configurations.

Given a CFG node $\mathcal{S}$, the set $livein(\mathcal{S})$ is the set of variables that are live on entry to the node, $liveout(\mathcal{S})$ is the set of variables live on exit from the node, $use(\mathcal{S})$ is the set of variables used by the node, and $def(\mathcal{S})$ is the set of variables defined by the node. We assume that any dead code has already been eliminated by a standard compiler optimization pass.

Selections. The statement $x := \phi_g(x_0, \ldots, x_{n-1})$ is placed at merge points in the CFG when x is live at the merge point, and there are multiple reaching definitions $x_0, \ldots, x_{n-1}$ through each distinct edge in the merge. More precisely, we insert a ϕ function for all variables $x \in liveout(\mathcal{S}) \cap def(\mathcal{S})$ where $\mathcal{S}$ is the selection statement (excluding the guard variable g).

The statement $x_0, \ldots, x_{n-1} := \phi_g^{-1}(x)$ is placed at the split point in the CFG when the branches of the selection contain uses of x, or if the branches define x and x is live at the split point. More precisely, we insert a ϕ^{-1} function for each variable $x \in livein(\mathcal{S}) \cap (use(\mathcal{S}) \cup (liveout(\mathcal{S}) \cap def(\mathcal{S})))$. If a ϕ^{-1} node is inserted at a split point but x is not live on exit from the selection, then $\mathbf{s}(x)$ nodes are placed at the end of each branch of the selection if x is not used in that branch. More precisely, we insert $\mathbf{s}(x)$ nodes at the end of the branch S_i when a ϕ^{-1} was placed at the split, and when $x \notin (liveout(\mathcal{S}) \cup use(S_i))$.

Variables are renamed in the obvious way. If there is a use of a variable in a branch of the selection, it is renamed to the appropriate version that was generated by the ϕ^{-1} function. Uses of a variable after a selection use the variable generated with the ϕ-function at the merge point, if applicable. These rules are the same as the ones used by Ananian [2]. After renaming, we have the following property that is crucial to the static token form: *given a variable, the control-flow condition that determines when it is defined is identical to the control-flow condition that determines when it is used.* It is precisely this condition that allows us to transform variables into tokens.

Repetitions. Repetitions pose a challenge because their CFG contains nodes that are both splits and merges, they have loop-carried dependencies, and because the guard variable introduces a special situation with respect to ϕ-function placement. The head of the loop can contain both ϕ and ϕ^{-1} functions, the body can contain ϕ^{-1} functions at the end, and the tail can contain ϕ functions. To illustrate this, consider the repetition

...
$x := init_x(...);$
$g := init_g(...);$
$*[g = 1 \longrightarrow S; x := f(x, ...); g := h(...)];$
...

The variable x could be used multiple times. The problem is that each use of x occurs under a different control-flow condition. Using explicit **if/then/goto** control-flow notation, we would like to write the repetition as

$$
\begin{array}{ll}
 & ... \\
 & x_0 := init_x(...); \\
 & g_0 := init_g(...); \\
 & \mathbf{if}\ \ g_0 = 0\ \ \mathbf{then}\ \ \mathbf{goto}\ \ T; \\
L: & x_2 := \phi_?(x_0, x_1); \\
 & S; x_1 := f(x_2, ...); g_1 := h(...); \\
 & \mathbf{if}\ \ g_1 = 1\ \ \mathbf{then}\ \ \mathbf{goto}\ \ L; \\
T: & x_3 := \phi_{g_0}(x_0, x_1); \\
 & ...
\end{array}
$$

but we are confronted with the problem of choosing the ϕ subscript at program point L. We would like to simply use

g_1, except for the problem of the initial value of g_1. We resolve this by using the $\mathbf{I}_0$ statement which, by definition, solves precisely this problem. We can now use g_1 as the subscript for the ϕ-function. The next issue is the fact that both x_0 and x_1 have definition conditions that do not match their use condition. We resolve this by the introduction of ϕ^{-1} functions at program points H and B as follows:

$$
\begin{array}{rl}
 & ... \\
 & x_0 := init_x(...); \\
 & g_0 := init_g(...); \\
H: & h_0, h_1 := \phi^{-1}_{g_0}(x_0); \\
 & \mathbf{if}\ \ g_0 = 0\ \ \mathbf{then\ \ goto}\ \ T; \\
 & \mathbf{I}_0(g_1); \\
L: & x_2 := \phi_{g_1}(h_1, h_4); \\
 & S; x_1 := f(x_2, ...); g_1 := h(...); \\
B: & h_3, h_4 := \phi^{-1}_{g_1}(x_1); \\
 & \mathbf{if}\ \ g_1 = 1\ \ \mathbf{then\ \ goto}\ \ L; \\
T: & x_3 := \phi_{g_0}(h_0, h_3); \\
 & ...
\end{array}
$$

In general, the statement $x := \phi_{g_1}(x_0, x_1)$ is placed at program point L, in the head of the repetition CFG, when the repetition contains uses of x and x is live at L. More precisely, we insert a ϕ function for all variables $x \in livein(\mathcal{R}) \cap use(\mathcal{R})$ where $\mathcal{R}$ is the repetition statement (excluding the guard variable g).

The statement $x := \phi_{g_0}(x_0, x_1)$ is placed at program point T, in the tail of the repetition CFG, when x is live at T and the repetition contains a reaching definition x_0 from the head of the CFG and a reaching definition x_1 from the body of the CFG. More precisely, we insert a ϕ function for all variables $x \in liveout(\mathcal{R}) \cap def(\mathcal{R})$.

The statement $x_0, x_1 := \phi^{-1}_{g_1}(x)$ is placed at program point B, in the body of the repetition CFG, when a ϕ function for x was placed at program points L or T. If $x \notin liveout(\mathcal{R})$, we insert an "$\mathbf{if}\ \ g_1 = 0\ \ \mathbf{then}\ \ \mathbf{s}(x_0)$" statement after the ϕ^{-1} function. Similarly, if $x \notin use(\mathcal{R})$ we insert an "$\mathbf{if}\ \ g_1 = 1\ \ \mathbf{then}\ \ \mathbf{s}(x_1)$" statement after the ϕ^{-1} function.

The statement $x_0, x_1 := \phi^{-1}_{g_0}(x)$ is placed at program point H, in the head of the repetition CFG, when a ϕ function for x was placed at program points L or T. If $x \notin liveout(\mathcal{R})$, we insert an "$\mathbf{if}\ \ g_0 = 0\ \ \mathbf{then}\ \ \mathbf{s}(x_0)$" statement after the ϕ^{-1} function. Similarly, if $x \notin use(\mathcal{R})$ we insert an "$\mathbf{if}\ \ g_0 = 1\ \ \mathbf{then}\ \ \mathbf{s}(x_1)$" statement after the ϕ^{-1} function.

Compiler Analyses. Transforming a canonical CHP program into ST form occurs during three compiler passes. First, a sequential control flow graph of the program is constructed and a liveness analysis is performed to determine at what program points each variable is live. Second, ϕ and ϕ^{-1} functions are inserted as necessary around selection and repetition statements. For programs with nested selection and repetition statements, it is necessary to iterate over these first two passes. This is required because the guard variables of the inner ϕ and ϕ^{-1} functions may change the liveness analysis of the outer statements and consequently require the insertion of additional ϕ and ϕ^{-1} functions. Finally, all variables are renamed to create a valid ST program representation.

4. Concurrent Dataflow Decomposition

A *concurrent dataflow* decomposition maps a sequential CHP program in ST form onto a concurrent dataflow graph. Since all variable definitions and variable uses have matching control-flow conditions in ST form, concurrent dataflow decomposition is a simple one-to-one mapping of program actions onto concurrent dataflow nodes. This decomposition maps variable definitions in the sequential program to asynchronous token producers in the concurrent graph and similarly, variable uses to asynchronous token consumers. The directed edges of the resulting dataflow graph represent the flow of tokens in the concurrent decomposition, which correspond to asynchronous channels in a physical pipeline implementation. Each dataflow graph node represents a concurrent asynchronous pipeline stage and is one of seven simple process types.

The seven types of concurrent dataflow nodes are shown in Figure 4 and their functionality is described below:

$$
\begin{array}{l}
Copy \equiv\ *[A?a; Z_0!a, ..., Z_{n-1}!a] \\
Function \equiv *[A_0?a_0, ..., A_{n-1}?a_{n-1}; Z!f(a_0, ..., a_{n-1})] \\
Split \equiv *[C?c, A?a; \\
\qquad [c = 0 \longrightarrow Z_0!a [] ... [] c = n-1 \longrightarrow Z_{n-1}!a]] \\
Merge \equiv *[C?c; \\
\qquad [c = 0 \longrightarrow A_0?a [] ... [] c = n-1 \longrightarrow A_{n-1}?a]; \\
\qquad Z!a] \\
Source \equiv\ *[Z!''constant''] \\
Sink \equiv\ *[A?a] \\
Initializer \equiv\ a := ''constant''; Z_0!a, ..., Z_{n-1}!a; \\
\qquad *[A?a; Z_0!a, \ldots, Z_{n-1}!a]
\end{array}
$$

We note that each concurrent dataflow node is simple enough to be implemented in a pipelined asynchronous FPGA [15, 16], as well as with custom fine-grain pipelined circuits. In addition to the seven dataflow nodes, there are several pseudo dataflow nodes (nodes that are not actually implemented in hardware) that serve as logical place holders in the dataflow graph and have no shapes surrounding their labels. Channel sends ($C!$) and channel receives ($C?$) are pseudo dataflow nodes and are connection place holders for environment processes.

We use the following steps to construct a concurrent dataflow graph from a sequential program in ST form:

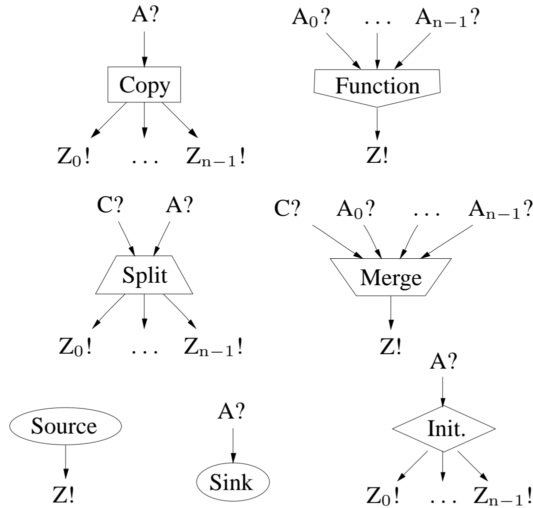

Figure 4. Concurrent dataflow nodes.

1. Map variables to copy nodes, functions and expressions to function nodes, ϕ functions to merge nodes, ϕ^{-1} functions to split nodes, $\mathbf{s}()$ actions to sink nodes, constant values to source nodes, and $\mathbf{I}()$ actions to initializer nodes.

2. Add directed edges to the graph, such that they follow the flow of data tokens from variable definitions to variable uses in the sequential program.

3. Replace (*extraneous*) copy nodes with pseudo nodes if the copy node has only one output edge.

4. For each initialization "channel send" action that appears before the main program loop, concatenate an initializer node before its corresponding "channel send" pseudo node.

Straight-Line Programs. Consider the following straight-line program and its ST form:

$$\begin{aligned} \mathit{Straight} &\equiv *[\ A?a; B?b; X!f(a,b); Y!f(b,1); \\ &\qquad D?a; Z!g(a)\ ] \\ \mathit{Straight}^{ST} &\equiv *[\ A?a_0; B?b_0; X!f(a_0,b_0); Y!f(b_0,1); \\ &\qquad D?a_1; Z!g(a_1)\ ] \end{aligned}$$

Figure 5(a) shows its initial concurrent dataflow graph after performing steps 1–2 of the concurrent dataflow decomposition and Figure 5(b) shows the finished dataflow graph after the extraneous copies have been removed. Observe that the sequential behavior between the first and second lines in the original program is completely removed when it is mapped to a concurrent dataflow graph, since there is no true data-dependence on the variable a after the program is converted to ST form. Functions (e.g., f and g) are treated as macro expansions for expressions and are not required to have unique names because each function "call" maps to its own function node. Note that source nodes attached to function nodes could be coalesced into the function node to decrease the size of the resulting pipeline.

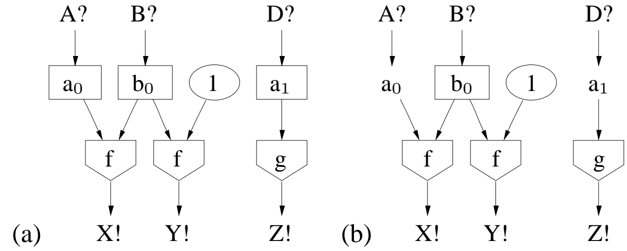

Figure 5. Straight-line program: (a) initial dataflow graph and (b) after removing extraneous copies.

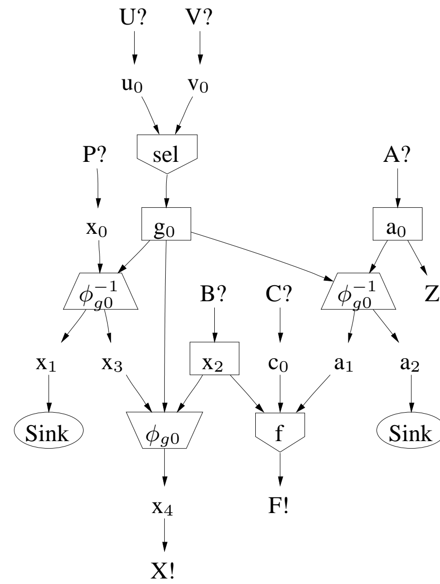

Figure 6. Program with selection statement.

Selections. Consider a program with selection statements

$$\begin{aligned} *[\ & U?u; V?v; A?a; P?x; \\ & [u \wedge v \longrightarrow B?x; C?c; F!f(a,x,c) \\ & \,[\!]\,\mathbf{else} \longrightarrow \mathbf{skip} \\ &];\ Z!a; X!x\ ] \end{aligned}$$

and its corresponding ST form.

$$\begin{aligned} *[\ & U?u_0; V?v_0; A?a_0; P?x_0; \\ & g_0 := \mathit{sel}(u_0, v_0); \\ & a_1, a_2 := \phi^{-1}_{g0}(a_0); \\ & x_1, x_3 := \phi^{-1}_{g0}(x_0); \\ & [g_0 = 0 \longrightarrow B?x_2; C?c_0; F!f(a_1, x_2, c_0); \mathbf{s}(x_1) \\ & \,[\!]\, g_0 = 1 \longrightarrow \mathbf{s}(a_2) \\ &];\ x_4 := \phi_{g0}(x_2, x_3); \\ & Z!a_0; X!x_4\ ] \end{aligned}$$

Figure 6 shows the concurrent dataflow graph for this program. Since $Z!a_0$ is not data dependent on the actions inside of the selection statement, it can execute concurrently with the selection statement, whereas in the original sequential program it executed after the selection statement.

Token Initializations. To illustrate how the token initializer dataflow node is used, consider a program that unconditionally toggles on which output channel it sends data.

$$\begin{aligned} \mathit{Toggle} \equiv\ & s := 0; X!1; \\ & *[A?x; [\neg s \longrightarrow X!x [\!] s \longrightarrow Y!x]; s := \neg s] \\ \mathit{Toggle}^{ST} \equiv\ & \mathbf{I}_0(s_0); X!1; \\ & *[A?x_0; x_1, x_2 := \phi_{s0}^{-1}(x_0); \\ & \quad [s_0 = 0 \longrightarrow X!x_1 \ [\!]\ s_0 = 1 \longrightarrow Y!x_2]; \\ & \quad s_0 := \neg s_0 \ ] \end{aligned}$$

Note that initializer nodes are generated for both $\mathbf{I}_0(s_0)$ and $X!1$ in the concurrent dataflow graph in Figure 7, and that the ϕ_{s0}^{-1} function uses the "initialized" value of s_0 instead of s_0 directly.

Repetitions. A program that outputs the first n positive integer numbers on channel X is listed below and its concurrent dataflow graph is shown in Figure 7.

$$\begin{aligned} \mathit{Loop} \equiv\ & *[N?n; G?g; x := 0; \\ & *[g = 1 \longrightarrow X!x; x := x + 1; g := (x \le n)]] \end{aligned}$$

$$\begin{aligned} \mathit{Loop}^{ST} \equiv\ & *[N?n_0; G?g_0; x_0 := 0; \\ H:\ & x_3, x_4 := \phi_{g_0}^{-1}(x_0); \\ & \mathbf{if}\ g_0 = 0\ \mathbf{then}\ \mathbf{s}(x_3); \\ & n_2, n_3 := \phi_{g_0}^{-1}(n_0); \\ & \mathbf{if}\ g_0 = 0\ \mathbf{then}\ \mathbf{s}(n_2); \\ & \mathbf{if}\ g_0 = 0\ \mathbf{then\ goto}\ T; \\ & \mathbf{I}_0(g_1); \\ L:\ & x_2 := \phi_{g_1}(x_4, x_6); \\ & n_1 := \phi_{g_1}(n_3, n_5); \\ & X!x_2; x_1 := x_2 + 1; g_1 := (x_1 \le n_1); \\ B:\ & x_5, x_6 := \phi_{g_1}^{-1}(x_1); \\ & \mathbf{if}\ g_1 = 0\ \mathbf{then}\ \mathbf{s}(x_5); \\ & n_4, n_5 := \phi_{g_1}^{-1}(n_1); \\ & \mathbf{if}\ g_1 = 0\ \mathbf{then}\ \mathbf{s}(n_4); \\ & \mathbf{if}\ g_1 = 1\ \mathbf{then\ goto}\ L \\ T:\ &] \end{aligned}$$

Observe that ϕ_{g_1} functions at program point L use the "initialized" value of g_1, whereas the $\phi_{g_1}^{-1}$ functions at program point B use g_1 directly. Also note that the node $x_3, x_4 := \phi_{g_0}^{-1}(x_0)$ is wasteful because $x_0 \equiv 0$, and so we could move the "zero-source" from x_0 to x_4 in the dataflow graph. This inefficiency is due to the restrictive syntax of CHP repetition statements and *not* due to a deficiency in ST form or concurrent dataflow decomposition. To prevent this problem, we would need to modify the high-level CHP syntax to allow the initialization of loop-carried variables at the head of repetition statements, instead of in the main loop.

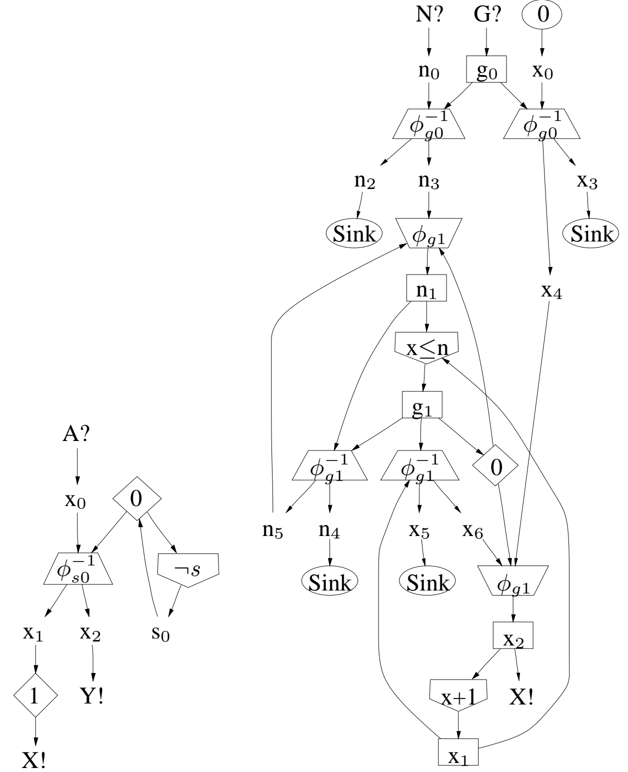

Figure 7. Toggle (left) and Loop (right) programs.

5. Correctness

In the previous section we have shown how a sequential program can be automatically transformed into a regular set of highly concurrent processes using concurrent dataflow decomposition. It is clear that such a decomposition increases the amount of *slack*, or number of pipeline stages, on channels that interface with the program's environment. However, this can change the relative order of actions that the environment observes on environment channels in the sequential and concurrent implementations of the program (although the absolute order of actions on individual channels is preserved). If a program can function correctly with an arbitrary amount of slack on its environment channels, then it is *locally slack elastic* [9].

In this section we present two theorems validating the correctness of concurrent dataflow decompositions. The first uses the properties of slack elastic systems [9] to describe when this decomposition can be safely applied and the second uses projection [10] to formally show that a concurrent dataflow graph is equivalent to its sequential ST program. The interested reader is referred to [9, 10] for further details on slack elasticity and projection.

Theorem 1 (correctness) *Let P be a sequential program and E be its environment, such that $P||E$ form a closed*

system. Concurrent dataflow decomposition is safe to use on P if the following conditions are satisfied: (1) no shared variables between P and E, (2) guards of selection statements in P and E are syntactically mutually exclusive, and (3) no probed channels in P and E.

Proof: Condition 1 is stipulated so that we can use the results derived in [9] for slack elastic systems. Conditions 2 and 3 are sufficient to apply Corollary 2 from [9], which says that P will be locally slack elastic. Since P is locally slack elastic we can safely apply concurrent dataflow decomposition to P. ∎

Theorem 1 holds for a large class of asynchronous systems [9], including an entire microprocessor design [12] (except for an arbiter in the cache system and a probed channel in the exception unit). Such programs can always be syntactically translated into ST form.

Theorem 2 (equivalence) *If a sequential program P satisfies the conditions stated in Theorem 1, then its concurrent dataflow graph is equivalent to P.*

Proof: (Sketch) Since P satisfies the conditions in Theorem 1, we can transform it into ST form. We then use projection [10] to construct a concurrent dataflow graph for P as follows:

1. Add channel communication actions for all conditional variables used in the ϕ and ϕ^{-1} functions.
2. Project out all selection branches in P, which is possible because ST form guarantees that branches have disjoint projection sets. Let P' be the resulting set of concurrent processes.
3. Apply *control duplication* [10] to obtain fresh copies of each guard variable in P'.
4. Project each variable in P' onto its own processes.
5. Decompose initialization actions into their own processes.

The resulting processes consist only of concurrent dataflow nodes and are equivalent to P by projection. ∎

Example: Consider the following simple program.

$$\begin{array}{l} *[G?g;\, Y?y; \\ \quad [g = 0 \longrightarrow y := y + 1 \,[]\, g = 1 \longrightarrow skip];\, Z!y \;\;] \end{array}$$

We first convert to ST form and then perform step 1.

$$\begin{array}{l} *[G?g_0;\, Y?y_0;\, y_1, y_2 := \phi_{g_0}^{-1}(y_0); \\ \quad [g_0 = 0 \longrightarrow Y_1!y_1 \,[]\, g_0 = 1 \longrightarrow Y_2!y_2]; \\ \quad [g_0 = 0 \longrightarrow Y_1?y_1;\, y_3 := y_1 + 1;\, Y_3!y_3 \\ \quad [] g_0 = 1 \longrightarrow skip \\ \quad];\, [g_0 = 0 \longrightarrow Y_3?y_3 \,[]\, g_0 = 0 \longrightarrow Y_2?y_2]; \\ \quad y_4 := \phi_{g_0}(y_3, y_2);\, Z!y_4 \;\;] \end{array}$$

Since this program is slack elastic, we assert that there is sufficient non-zero slack on channels Y_1, Y_2 and Y_3 such that the above sequential program will not deadlock. Applying projection to the variables contained in the original selection branches, we obtain:

$$\begin{array}{l} *[G?g_0;\, Y?y_0;\, y_1, y_2 := \phi_{g_0}^{-1}(y_0); \\ \quad [g_0 = 0 \longrightarrow Y_1!y_1 \,[]\, g_0 = 1 \longrightarrow Y_2!y_2]; \\ \quad [g_0 = 0 \longrightarrow skip \,[]\, g_0 = 1 \longrightarrow skip]; \\ \quad [g_0 = 0 \longrightarrow Y_3?y_3 \,[]\, g_0 = 1 \longrightarrow Y_2?y_2]; \\ \quad y_4 := \phi_{g_0}(y_3, y_2);\, Z!y_4 \;\;] \\ \|\, *[Y_1?y_1;\, y_3 := y_1 + 1;\, Y_3!y_3] \end{array}$$

We remove the empty selection statement, merge the ϕ and ϕ^{-1} functions with the remaining selections, and use control duplication to copy g_0.

$$\begin{array}{l} *[G?g_0;\, (G_1!g_0 \,\|\, G_1?g_1);\, (G_2!g_0 \,\|\, G_2?g_2);\, Y?y_0; \\ \quad [g_1 = 0 \longrightarrow y_1 := y_0;\, Y_1!y_1 \\ \quad\; [] g_1 = 1 \longrightarrow y_2 := y_0;\, Y_2!y_2]; \\ \quad [g_2 = 0 \longrightarrow Y_3?y_3;\, y_4 := y_3 \\ \quad\; [] g_2 = 1 \longrightarrow Y_2?y_2;\, y_4 := y_2];\, Z!y_4 \;\;] \\ \|\, *[Y_1?y_1;\, y_3 := y_1 + 1;\, Y_3!y_3] \end{array}$$

Finally, we project each variable into its own process.

$$\begin{array}{l} *[G?g_0;\, G_1!g_0, G_2!g_0] \\ \|\, *[G_1?g_1, Y?y_0;\, [g_1 = 0 \longrightarrow y_1 := y_0;\, Y_1!y_1 \\ \qquad\qquad\qquad\quad [] g_1 = 1 \longrightarrow y_2 := y_0;\, Y_2!y_2]] \\ \|\, *[G_2?g_2;\, [g_2 = 0 \longrightarrow Y_3?y_3;\, y_4 := y_3 \\ \qquad\qquad\quad [] g_2 = 1 \longrightarrow Y_2?y_2;\, y_4 := y_2];\, Z!y_4] \\ \|\, *[Y_1?y_1;\, y_3 := y_1 + 1;\, Y_3!y_3] \end{array}$$

Observe that this decomposed system is composed entirely of concurrent dataflow nodes and is equivalent to the system generated by concurrent dataflow decomposition.

6. Implementation and Applications

The pipeline synthesis framework presented in this paper has been implemented using *Cyclone*, a type-safe "C-like" programming language [8]. At the time of this writing, the core functionality of this hardware compiler, namely the sequential compiler analysis and concurrent dataflow decomposition for programs with selections, is complete and occupies approximately 3000 lines of code (not including parsing and AST construction). Although we are continuing work on implementing the analysis for internal repetitions and multiple channel actions, we have a working CHP compiler that can automatically decompose relatively complex sequential programs. Static token form and concurrent dataflow decomposition are sufficiently general such that other high-level languages could be easily adapted to substitute for CHP as the front-end language in our pipeline compiler.

Table 1. Technology mapping examples.

Program	High-level pipeline synthesis			FPGA technology mapping		
	CHP actions	ST actions	dataflow nodes	bit-level nodes*	logic blocks	throughput (MHz)
Fibonacci (32-bit)	8	8	5	160	96	668
Writeback Unit	17	25	19	63	54	658
Loop (4-bit)	6	15	16	48	32	189

*not including source, sink, and initializer nodes

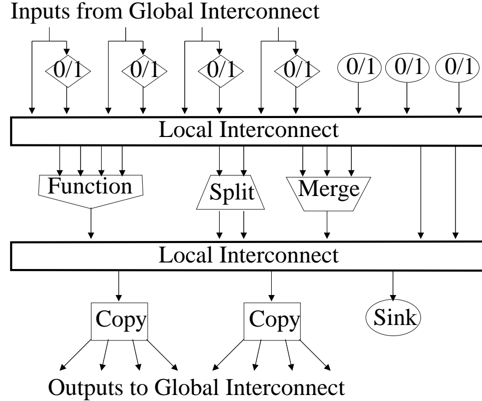

Figure 8. Concurrent dataflow graph representation of a pipelined asynchronous FPGA logic block.

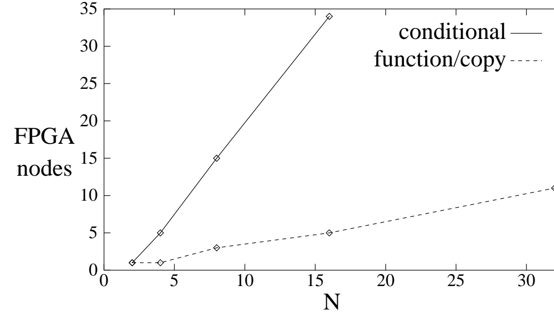

Figure 9. Asynchronous FPGA scaling trends for implementing N-input function, N-way conditional, and N-way copy (bit-level) dataflow nodes.

We have found that this compiler generates dataflow pipelines that are equivalent to dataflow pipelines that we previously synthesized manually. However, the quality of these pipelines is sometimes limited by the quality of the sequential CHP, especially for programs with selection statements. For example, consider the following two equivalent programs:

$$
\begin{array}{ll}
*[\,G?g;\,A?a;\,B?b; & *[\,G?g;\,A?a;\,B?b; \\
\quad [\,g=0 \longrightarrow z:=a & \quad z:=\neg g \wedge (a) \\
\quad [\!]\,g=1 \longrightarrow z:=b & \qquad\quad \vee g \wedge (b); \\
\quad];\,Z!z\ ] & \quad Z!z\ ]
\end{array}
$$

The program on the left maps to six dataflow nodes, whereas the equivalent program on the right requires only one dataflow node. While an experienced designer may recognize that the selection statement in the left program is not semantically necessary, our current compiler does not yet implement such semantic optimizations.

Asynchronous FPGAs. Recent work has investigated pipelined asynchronous FPGA architectures [15, 16] that efficiently implement programmable computation nodes similar to those used in concurrent dataflow graphs. Figure 8 shows a concurrent dataflow graph representation of a typical pipelined asynchronous logic block used in these FPGA architectures. The asynchronous logic block uses bit-level dual-rail channels and contains a 4-input function node, a two-way conditional split, a two-way conditional merge, and four-way output copy nodes. In addition, a logic block has built-in initializer nodes, constant sources, and sink nodes. Given the functionality of these pipelined asynchronous FPGA architectures, it is clear that they can implement arbitrarily complex concurrent dataflow graphs.

However, before a concurrent dataflow graph can be implemented on an asynchronous FPGA, the graph's multi-bit dataflow nodes must first be mapped to bit-level nodes that are equal in size to the FPGA's nodes. Observe that in an asynchronous FPGA we get multi-bit source, sink, and initializer nodes essentially for free since they are built into each bit-level logic block. This is not true for function, conditional, and copy nodes because they have limited fanin/fanout in an FPGA node, but infinite fanin/fanout in a dataflow node. Using Figure 9 we can estimate the number of bit-level FPGA nodes required to implement an N-input function, N-way conditional, or N-way copy dataflow node.

Table 1 gives statistics for several asynchronous programs that were synthesized using our pipeline compiler and technology mapped to an asynchronous FPGA. *Fibonacci* is a straight-line program that sequentially generates all of the Fibonacci numbers that can be represented by 32-bit integers. The *MiniMIPS writeback unit* [10] is a more typical asynchronous hardware process that contains several selection statements and state variables. The *Loop* program is a 4-bit implementation of the concurrent dataflow graph shown in Figure 7. Our target FPGA is similar to the

Table 2. Comparing asynchronous pipeline synthesis methods for $*[A?a; b := \neg a; [\neg b \rightarrow B!b[]b \rightarrow X?x; Y!y(x)]]$.

Decomposition Method	Intermediate Form	Decomposed Processes
Data-Driven Decomposition [19]	$*[A?a; b := \neg a;$ $[\neg b \longrightarrow B!b$ $[] b \longrightarrow X?x;\ Y!y(x)]]$	$*[A?a; b := \neg a; B_0!b, B_1!b]$ $\|\| *[B_0?b; [\neg b \longrightarrow \textbf{skip} [] b \longrightarrow X?x;\ Y!y(x)]]$ $\|\| *[B_1?b; [\neg b \longrightarrow B!b [] b \longrightarrow \textbf{skip}]]$
Concurrent Dataflow Decomposition	$*[A?a_0; b_0 := \neg a_0;$ $b_1, b_2 := \phi_{b0}^{-1}(b_0);$ $[b_0 = 0 \longrightarrow B!b_1$ $[] b_0 = 1 \longrightarrow X?x_0;\ Y!y(x_0); \mathbf{s}(b_2)$ $]]$	$*[A?a; B_0!\neg a]$ $\|\| *[B_0?b; B_b!b, B_c!b]$ $\|\| *[B_c?c, B_b?b; [c = 0 \longrightarrow B!b [] c = 1 \longrightarrow B_1!b]]$ $\|\| *[B_1?b]$ $\|\| *[X?x;\ Y!y(x)]$

logic block shown in Figure 8, contains heavily pipelined interconnects, and has a peak operating frequency of 700 MHz [16]. We automatically placed and routed the resulting technology-mapped dataflow graphs onto this FPGA architecture. Throughput values were obtained from a detailed switch-level asynchronous FPGA simulator, which used delay values extracted from an asynchronous FPGA laid out in a typical TSMC $0.18\mu m$ process. The Loop program is slower than the other two benchmarks because its throughput is limited by the update computation for g_1.

Custom Logic. Custom logic presents a much larger design space than an FPGA, and there are many circuit optimizations available to improve the performance of the concurrent dataflow graph when it is implemented with full-custom circuits. For example, we can coalesce copy nodes with other dataflow nodes to reduce pipeline latency, coalesce sink nodes with split nodes to decrease energy consumption, and slack match [12] the edges of the dataflow graph to improve throughput.

Program Visualization. A side benefit of automated concurrent dataflow decomposition is that it lets an asynchronous logic designer visualize the maximum potential concurrency hidden in a sequential program specification. Since each concurrent dataflow node implements a simple well understood function, by examining the program's concurrent dataflow graph, a designer can quickly analyze how program actions interact with each other.

7. Related Work

Concurrent dataflow graphs are functionally similar to dataflow program graphs used in a software compiler for the MIT tagged-token dataflow computer architecture [17]. In this compiler, program graphs were used for high-level program transformations and were then compiled into machine code for a tagged-token hardware architecture that had very little resemblance to the original program graph. However, we use concurrent dataflow graphs to describe the underlying concurrent hardware directly, and not only for sequential program analyses.

High-level language compilers for clocked hardware traditionally limit the amount of pipelining in the resulting system. A sequential program is usually synthesized as combinational logic, with pipelining only introduced across loop iterations and at procedure calls (e.g., [6]). Since most synchronous systems are not slack elastic, it is not safe for them to introduce concurrency and pipelining as freely as our compiler does when it produces a concurrent dataflow graph. For example, the Handel-C synchronous compiler [14], which allows a programmer to use CSP [7] concurrency and channel communication constructs, assumes most of the parallelism is explicitly specified in the source program instead of determined by the compiler. The Handel-C compiler also makes circuit assumptions about statement execution delays in terms of clock cycles and introduces a high-level `Delay` statement, which waits for one clock cycle.

The *data-driven decomposition* method [19] is the only other known asynchronous synthesis framework for automatically decomposing a slack-elastic program into fine-grain concurrent processes. This method projects each high-level program variable into its own process. The disadvantage of this method for FPGA synthesis is that it produces processes that can be more coarse-grain than our dataflow nodes and in addition, they are not regular in their functionality. In the worst case, a data-driven decomposition requires a different circuit to be designed for each process, limiting the applicability of this method to full-custom asynchronous designs. Furthermore, the data-driven decomposition method is less practical for some high-level program decompositions because it uses a *dynamic single assignment* (DSA) form[1] that, by definition, precludes the synthesis of nested loops (without additional AST modifications that would syntactically eliminate the nesting).

To illustrate the differences between data-driven decomposition and concurrent dataflow decomposition, consider the following example:

$$*[A?a; b := \neg a; [\neg b \longrightarrow B!b [] b \longrightarrow X?x;\ Y!y(x)]]$$

[1] DSA form is equivalent to SSA form, with ϕ functions replaced by explicit assignment statements at the end of selection branches.

Table 2 shows the intermediate forms and final decomposed processes for both pipeline synthesis methods. Since the value communicated on Y is a function of the value received on X and does not depend on a or b, the statements $X?x$ and $Y!y(x)$ should be projected out of the selection branch to maximize concurrency in the decomposed system [10]. This happens naturally during concurrent dataflow decomposition, but does not occur during data-driven decomposition because the selection guard b is included in the dependency set for x [21]. In contrast, since our asynchronous FPGA architecture does not have explicit conditional outputs, our method requires a fixed format for split logic that necessitates an extra copy of variable b as well as an explicit sink. Observe that concurrent dataflow decomposition can be applied to the processes produced by data-driven decomposition to yield a system that is similar to the one originally generated by our method. In addition, the nodes produced by our method can be clustered using the techniques outlined by Wong to more closely match the pipeline stages used in full-custom implementations [20].

8. Summary

We presented an automated decomposition method for the high-level synthesis of finely pipelined asynchronous systems. We introduced a new sequential compiler analysis that can transform a sequential program into a regular set of highly concurrent processes. We designed a pipeline compiler and showed that it can be used to synthesize logic for pipelined asynchronous FPGAs.

Acknowledgments

The research described in this paper was supported in part by the Multidisciplinary University Research Initiative (MURI) under the Office of Naval Research Contract N00014-00-1-0564, and in part by an NSF CAREER award under contract CCR 9984299. John Teifel was supported in part by an NSF Graduate Research Fellowship.

A. Summary of CHP Notation

The CHP notation we use is based on Hoare's CSP [7]. A full description of CHP and its semantics can be found in [11]. What follows is a short and informal description.

- 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 $a := false$.
- Selection: $[G1 \rightarrow S1 \; [] \; ... \; [] \; Gn \rightarrow Sn]$, where Gi's are boolean expressions (guards) and Si's 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 notation $[G]$ is short-hand for $[G \rightarrow skip]$, and denotes waiting for the predicate G to become true. If the guards are not mutually exclusive, we use the vertical bar "|" instead of "[]."
- Repetition: $*[G1 \rightarrow S1 \; [] \; ... \; [] \; Gn \rightarrow Sn]$. The execution of this command corresponds to choosing one of the $true$ guards and executing the corresponding statement, repeating this until all guards evaluate to $false$. The notation $*[S]$ is short-hand for $*[true \rightarrow S]$.
- 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 expression $\overline{X}$ is $true$ iff a communication over channel X can complete without suspending.
- Sequential Composition: $S; T$
- Parallel Composition: $S \parallel T$ or S, T.
- Simultaneous Composition: $S \bullet T$ both S and T are communication actions and they complete simultaneously.

B. Compiler Terminology

What follows is a brief summary of the compiler terms that we used in this paper. Muchnick provides a complete introduction to modern compiler techniques, including static single assignment form [13].

- Abstract Syntax Tree (AST): Compiler data structure that stores a parsed program in a source-syntax neutral representation.
- Control Flow Graph (CFG): Compiler data structure that stores a static representation of a program, and shows all alternatives of control flow. Nodes in the graph are basic blocks, straight-line pieces of code without any branches, and directed edges represent branches in control flow. A *def* is a CFG node that defines a variable. A *use* is a CFG node that uses a variable.
- Def-use chain: The control flow paths that connect a variable *def* to all of its potential *uses*.
- Use-def chains: The control flow paths that connect a variable *use* to all of its potential *defs*.

- Reaching definition: A definition reaches a CFG node when there is a control flow path from the variable's definition to that CFG node.

- Live variable: A variable is live on a CFG edge if there is a path from that edge to a use of the variable that does not go through any def. Liveness analysis computes live variable sets along each edge of a CFG.

C. Multiple Channel Actions

In static token form, channel actions must be treated as definitions. However, we cannot simply rename channels to remove multiple channel actions because this changes the program's behavior in a non-trivial manner and modifies the port interface that is expected by its environment. To solve this problem we introduce "channel sequencer" processes as illustrated in the following example:

$$T \equiv *[\ldots A? \ldots Z! \ldots A? \ldots G?g; \\ [g = 0 \longrightarrow Z! [] g = 1 \longrightarrow skip]; \ldots]$$

To remove multiple channel actions we rename channels:

$$T' \equiv *[\ldots A_0? \ldots Z_0! \ldots A_1? \ldots G?g; G'!g; \\ [g = 0 \longrightarrow Z_1! [] g = 1 \longrightarrow skip]; \ldots]$$

and introduce new "channel sequencer" processes:

$$A_{seq} \equiv s := 0; *[A?a; [s = 0 \longrightarrow A_0!a; s := 1 \\ [] s = 1 \longrightarrow A_1!a; s := 0 \ ]]$$

$$Z_{seq} \equiv s := 0; \\ *[[s = 0 \longrightarrow Z_0?a; out := 1; s := 1 \\ [] s = 1 \longrightarrow G'?g; [g = 0 \longrightarrow Z_1?a; out := 1 \\ [] g = 1 \longrightarrow out := 0]; \\ s := 0 \\ ]; [out = 0 \longrightarrow skip \ [] \ out = 1 \longrightarrow \ Z!a]]$$

such that $T \equiv A_{seq} \parallel T' \parallel Z_{seq}$. Since the size of these channel sequencers can be on the order of the original process, it is best to also syntactically coalesce multiple channel actions where possible (e.g., by merging branches with channel actions appearing in different branches of the same selection statement).

References

[1] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting Equality of Values in Programs. *Proc. of the 15th ACM Symposium on Principles of Programming Languages*, 1988.

[2] C. Scott Ananian. *The Static Single Information Form*. Master's thesis, Massachusetts Institute of Technology, 1999.

[3] Steven M. Burns and Alain J. Martin. Syntax-directed Translation of Concurrent Programs into Self-timed Circuits. *Proc. Fifth MIT Conference on Advanced Research in VLSI*, 1988.

[4] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, **12**(4):451-490, October 1991.

[5] Doug Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, **45**(1):12-18, 2002.

[6] David Galloway. The Transmogrifier C Hardware Description Language and Compiler for FPGAs. *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, 1995.

[7] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, **21**(8):666–677, 1978

[8] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. *USENIX Annual Technical Conference*, June 2002.

[9] Rajit Manohar and Alain J. Martin. Slack Elasticity in Concurrent Computing. *Proc. of the Fourth International Conference on the Mathematics of Program Construction*, 1998.

[10] Rajit Manohar, Tak-Kwan Lee, and Alain J. Martin. Projection: A Synthesis Technique for Concurrent Systems. *Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1999.

[11] Alain J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, **1**(4), 1986.

[12] A.J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. V. Cummings, and T.K. Lee. The Design of an Asynchronous MIPS R3000. *Proceedings of the 17th Conference on Advanced Research in VLSI*, September 1997.

[13] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[14] Ian Page. Constructing Hardware-Software Systems from a Single Description. *Journal of VLSI Signal Processing*, **12**(1), 1996.

[15] John Teifel and Rajit Manohar. Programmable Asynchronous Pipeline Arrays. *Proceedings of the 13th International Conference on Field Programmable Logic and Applications*, September 2003.

[16] John Teifel and Rajit Manohar. Highly Pipelined Asynchronous FPGAs. *Proceedings of the 12th ACM International Symposium on Field-Programmable Gate Arrays*, February 2004.

[17] Kenneth R. Traub. *A Compiler for the MIT Tagged-Token Dataflow Architecture.* M.S. Thesis, Massachusetts Institute of Technology, 1986.

[18] Kees van Berkel. *Handshake Circuits : An Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1994.

[19] Catherine G. Wong and Alain J. Martin. Data-Driven Process Decomposition for Circuit Synthesis. *Proc. of the IEEE Conference on Electronic Circuits and Systems*, 2001.

[20] Catherine G. Wong and Alain J. Martin. High-Level Synthesis of Asynchronous Systems by Data-Driven Decomposition. *Proc. of the 40th Design Automation Conference*, 2003.

[21] Catherine G. Wong. Personal Communication, August 2003.