Abstract

A System for Optimization and Synthesis of Asynchronous Circuits

Karthi Srinivasan

2026

Asynchronous Very-Large-Scale-Integration (VLSI) integrated circuits have several advantages over conventional synchronous circuits, such as lower energy consumption, input-dependent performance, robustness against manufacturing variations, etc. However, the absence of a high-quality logic synthesis system that operates on a sufficiently high-level description encumbers designers and limits the complexity of realizable designs. The relative performance of automatically synthesized circuits vis-a-vis synchronous designs has also proven a barrier.

In this thesis, we propose a novel logic synthesis technique for asynchronous circuits from a description at the level of a system of concurrent message-passing programs. The language we use for this purpose is Communicating Hardware Processes (CHP). Experiments show that the circuits synthesized by our technique are of the same quality, in terms of energy, delay and area, as those hand-designed by experts for designs where it is feasible to do so, and improve significantly on the state-of-the-art technique for those where it is not. Further, comparisons with open-source synchronous logic synthesis tools show parity in terms of area, delay and energy.

We also demonstrate a framework to unify and automate several different optimization techniques for CHP. In particular, we show how pipelining, projection, slack matching, dataflow decomposition etc. can be formulated as a unified optimization on the underlying timing graph of the circuit. In contrast to prior work, our technique allows flexibility in the degree of pipelining. Experiments show significant improvement in throughput for complex designs at the cost of increased area and energy consumption.

A System for Optimization and Synthesis of Asynchronous Circuits

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Karthi Srinivasan

Dissertation Director: Prof. Rajit Manohar

May 2026

Acknowledgments

I thank my advisor, Prof. Rajit Manohar, for his invaluable guidance and mentorship, for teaching me about concurrent systems and asynchronous circuits, and for always entertaining my questions about a wide variety of topics. He has always encouraged me to explore several avenues and think deeply about research problems.

I thank the members of my committee, Prof. Abhishek Bhattacherjee, Prof. Rajit Manohar and Prof. Linghao Song, for their feedback on this dissertation.

I thank Thomas Jagielski, Ruslan Dashkin, Congyang Li, Xiayuan Wen, Mattia Vezzoli, Jakob Jordan, Prafull Purohit, Ole Richter, Fabian Posch, Darja Nonaca, Siva Nalabothu, Kameron Gano, Luis Zuniga and other members of the AVLSI Lab for the insightful conversations that have shaped this work.

I thank my friends at Yale and beyond for their encouragement over the last few years. Thanks to Vallabh Ramakanth, Aditya Sundararajan, Vighnesh Natarajan, Suhas Pai, Gautham Umasankar and others for discussions that have kept me excited about science, math, research and the quest for knowledge.

I thank my mother for always encouraging me to follow my dreams and be the best version of myself.

Finally and most importantly, I thank my wife for her unwavering faith in me, especially when I had none in myself.

To those who strive to make the fruits of science free for all mankind.

Glossary

ACT	:	Asynchronous Circuit Tools/Toolkit
AST	:	Abstract Syntax Tree
ASIC	:	Application-Specific Integrated Circuit
CAD	:	Computer-Aided Design
CHP	:	Communicating Hardware Processes
DAG	:	Directed Acyclic Graph
DDG	:	Dependence-based Decomposition Graph
DI	:	Delay Insensitive
EDA	:	Electronic Design Automation
ER	:	Event-Rule
FIFO	:	First-In First-Out
FPGA	:	Field Programmable Gate Array
GND	:	Ground (low voltage in circuits)
HLS	:	High-Level Synthesis
HSE	:	Handshaking Expansions
I/O	:	Input/Output
ITB	:	Initial Token Buffer
LCD	:	Loop-Carried Dependency
NDS	:	Non-Deterministic Selection
PAA	:	Passive-Active-Active
PPA	:	Passive-Passive-Active
PRS	:	Production Rule Set
QDI	:	Quasi-Delay Insensitive
RER	:	Repetitive Event-Rule
RTL	:	Register-Transfer Level
SCC	:	Strongly-Connected Component
SDT	:	Syntax-Directed Translation
SI	:	Speed Independent
SPICE	:	Simulation Program with Integrated Circuit Emphasis
SSA	:	Static Single Assignment
SSI	:	Static Single Information
STF	:	Static Token Form
TG	:	Timing Graph
VDD	:	Voltage Drain (high voltage in circuits)
VLSI	:	Very Large Scale Integration
WCC	:	Weakly-Connected Component

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The modern world is inextricably linked to computing and the physical substrate that enables it — the integrated circuit. From the few dozen transistors in the 1960s to the billions in the 2020s, integrated circuits have seen rapid advancements in complexity [1]. Naturally, the process of designing these chips has also become more complex. As the number of electrical signals in the circuit grows to such proportions, it becomes necessary to enforce a paradigm that forms the foundation upon which the design is based. For historical reasons, the chosen one was the synchronous circuit paradigm which designates a master signal, known as the global clock, that defines the time reference for all components in the system. All computation in the circuit is synchronized by this signal.

The synchronous paradigm has granted designers simplicity of design, optimization and verification. However, over the course of time, this simplifying assumption has proven to be a bottleneck, for two principal reasons. Firstly, as speeds at which circuits operate have grown, it has become a major challenge to ensure that all parts of the integrated circuit receive the clock signal without relative delay (known as skew). Secondly, the network of logic gates that distribute the clock signal in a way that minimizes skew can consume a significant fraction of the total power expended by the chip.

Several solutions to these problems have been explored. Clock-gating [2, 3] prevents

the clock signal from causing switching activity in places where it is not needed, based on the state of the system. Globally-asynchronous locally-synchronous circuits [4, 5] that have multiple independent clock domains allow weakening of the global synchronization requirement by having smaller synchronized regions that communicate with each other over interfaces that ensure clean data transmission across the synchronization boundary.

Asynchronous circuits (also known as clockless or self-timed circuits) [6–8] take this idea of having independent clock domains to its logical extreme, by having each individual state machine determine its own rate of advancement through time, a function which the global clock typically performs, via a controller circuit. These individual circuits communicate with each other over wire bundles (known as channels) where the transmission is synchronized using a handshake protocol. Although asynchronous circuits have existed for several decades, the greater design complexity and area overhead have precluded widespread adoption in the semiconductor industry. The dominance of the synchronous paradigm has also created a positive feedback loop that has resulted in EDA tool support further evolving to be more compatible with it.

In the quest to make asynchronous circuit design competitive with its synchronous counterpart, there have been several projects over the years to develop a commercial-grade EDA tool flow [9–12]. The most recent has been the ACT EDA tools project [13], which is capable of supporting many different families of asynchronous circuits from logic synthesis all the way down to GDSII.

Our contribution in this work is two-fold. First, we provide a solution to the problem of synthesizing a high-quality asynchronous circuit from the high-level behavioral description language in ACT. Second, we provide techniques to automatically optimize the performance of these circuits. Taken together, these techniques (and their corresponding software implementations) allow designers to obtain circuits that are as performant as hand-optimized ones without expending the effort.

Chapter 2

Background

2.1 Communicating Hardware Processes

CHP is a hardware description language used to describe clockless circuits derived from C.A.R. Hoare's Communicating Sequential Processes (CSP) [14]. In CHP, circuits are described at an abstract level as processes, each of which has some internal state, represented by variables, and a program that determines the sequence of actions that this process performs. Processes do not directly have access to internal variables of other processes, but rather communicate values across channels. Channels are unidirectional and ownership of the sending and receiving sides of a channel are fixed. If two processes need bidirectional communication then two channels, with their relative orientations flipped, are needed.

A CHP program, which specifies the behavior of a process can consist of the following constructs:

- Skip: $skip$ is an elementary action that does nothing and continues to the next action.
- Assignment: $x := e$ is an elementary action that sets the variable x to the value e, where e is an expression. Since we often set Boolean variables to $true$ or $false$, we introduce the short-hand notation $x\uparrow$ and $x\downarrow$ to correspond to $x := true$ and $x := false$ respectively.

- Send: $C!e$ is an elementary action that sends the value of the expression e over the channel C. $C!$ is a dataless send, where no value is sent over the channel, but a communication is performed nevertheless for synchronization purposes. Channels do not buffer values; hence, a send will block until the corresponding receive is attempted, and vice versa.

- Receive: $C?x$ is an elementary action that receives a value over the channel C and stores it in the variable x. $C?$ is a dataless receive, where the received value is ignored, and only the synchronization is performed.

- Channel Access: C is a shorthand used in some places in this work, for either $C!$ or $C?$ when it is not relevant to differentiate between the sending and receiving ends of the channel.

- Channel Probe: $\overline{C}$ is a Boolean used to determine if the channel has a pending action to be completed. Both the sending and the receiving end of a channel can be probed. If the sender (receiver) probe is true, that means that the receiving (sending) end is blocked waiting for the sender (receiver).

- Sequential Composition: $S; T$ executes S followed by T, where S and T are any CHP programs.

- Parallel Composition: $S \parallel T$ or S, T executes the programs S and T in parallel, i.e. they may be executed in any order.

- Deterministic Selection: $[G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n]$ where G_i, known as guards, are Boolean-valued expressions and S_i are CHP programs, known as branches. The selection waits until one of the guards, G_i, evaluates to $true$, then executes the corresponding program, S_i. This construct is similar to a switch-case statement in several programming languages. The guards must be pairwise mutually exclusive,

i.e. at most one of them can be true. In any selection with two or more branches, the last guard can also be an $else$, which is simply a shorthand for writing $\neg(\vee_i G_i)$, the negation of the disjunction of all the previous guards. The notation $[G]$ is shorthand for $[G \rightarrow skip]$, which corresponds to waiting for G to become true, then proceeding.

- Non-Deterministic Selection: $[|\, G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n \,|]$ is identical to the deterministic selection except that the guards do not have to be mutually exclusive. If two or more evaluate to $true$ simultaneously, then one is picked arbitrarily (not necessarily at random). In a circuit, this choice is implemented by a collection of arbiters. When two or more guards evaluate to $true$ simultaneously, it can cause a metastable state in the arbiter. This metastable state then resolves non-deterministically, giving the grant to one of the branches of the selection statement. Therefore, the digital model of this selection statement is also non-deterministic in such a condition.

- Loop/Repetition: $*[G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n]$ is similar to the deterministic selection statement. However, once a branch has completed execution, the guards are re-evaluated. If one is true (guards must be pairwise mutually exclusive), then the corresponding branch is executed, and the process repeats until none are true. When this occurs, the loop terminates. Note that a loop may terminate without executing any branch even once, if all guards are false the first time it is reached. This construct is similar to a while loop, with the difference of having several possible branches instead of one. $*[S]$ is shorthand for $*[true \rightarrow S]$, which corresponds to an infinite loop – one that perpetually executes S.

- Do-Loop: $*[S \longleftarrow G]$ is similar to a loop, but first executes S then evaluates the guard G. If it evaluates to $true$, then the process repeats, until G evaluates to $false$. Do-loops may only have a single branch. This construct is similar to a do-while loop in programming languages.

2.2 Examples

This is a CHP program that receives two inputs, computes the greatest common divisor of the two using Euclid's algorithm and sends out the result:

$$
\begin{array}{l}
*[X?x,\, Y?y; \\
\quad *[x > y \longrightarrow x := x - y \\
\quad\quad [] y > x \longrightarrow y := y - x \\
\quad\quad];\, O!x]
\end{array}
$$

In general, a circuit is specified as a collection of processes operating in parallel. This is a CHP program for a three-stage pipeline:

$$*[A?a; B!f(a)] \parallel *[B?b; C!g(b)] \parallel *[C?c; D!h(c)]$$

This is a CHP program for a process that merges the values on two input channels into one, non-deterministically:

$$
\begin{array}{l}
*[\ [|\overline{L}_1 \longrightarrow L_1?x \\
\quad\quad [] \overline{L}_2 \longrightarrow L_2?x \\
\quad\quad |];\ \ R!x\ ]
\end{array}
$$

2.3 Handshakes and C-Elements

In order to enable ease of description of circuit behavior at a lower level, we use a language closely related to CHP, known as handshaking expansions (HSE). HSE is identical to CHP, with the additional constraint that all variables are Boolean-valued, which makes it a natural choice for circuit-level descriptions.

In order to provide some context for the synthesis of CHP to circuits, we first describe some primitives. Firstly, the abstract channels between processes need to be translated

to operations on wires/signals. Each channel in our synthesis is implemented with two wires, the request and the acknowledge. The process at each end of the channel senses one wire and drives the other, according to the four-phase handshake protocol. The process that drives the request of a particular channel is known as the active side and the side that drives the acknowledge is known as the passive side.

The protocol is as follows:

1. Initially both wires are low.
2. To initiate a communication, the active side sets the request wire high.
3. When the passive side wishes to acknowledge, it sets the acknowledge wire high.
4. To reset, after receiving the acknowledge, the active side pulls the request wire low.
5. In response to request being lowered, the passive side pulls the acknowledge low.
6. The channel is now back in its original state and another handshake can be performed.

In our notation, this is written succinctly as:

$$\begin{aligned} &Active: \quad r\uparrow;\ \ [a];\ \ r\downarrow;\ \ [\neg a] \\ &Passive: [r];\ \ a\uparrow;\ \ [\neg r];\ \ a\downarrow \end{aligned}$$

where $[x]$ denotes waiting for x to be true. Note that the four-phase handshake is not the minimal protocol for communication; there also exist two-phase handshakes where the first two actions by themselves (on each side) constitute an entire handshake synchronization. Instead of using the lowering to just reset the channel variables, they can be used to perform another handshake synchronization.

Next, we turn to the C-element half-buffer circuit [15]. A non-inverting C-element behaves as follows:

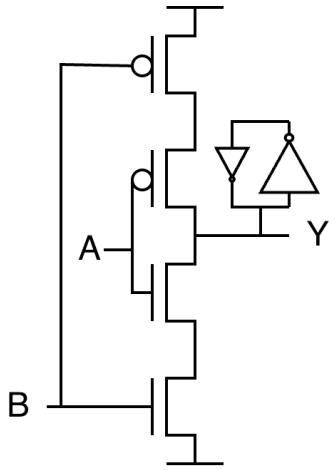


Figure 2.1: Transistor-level implementation of a 2-input inverting C-element.

1. If all inputs are low, then the output is low.

2. If all inputs are high, then the output is high.

3. In all other cases, it holds state (the output for the previous inputs).

Inverting C-elements are identical, with the exception of the output being inverted with respect to the non-inverting C-element. Fig. 2.1 shows the CMOS implementation of an inverting 2-input C-element, with a weak feedback staticizer, also known as a keeper. This appropriately sized cross-coupled inverter pair (smaller size in the figure denotes a weak inverter) maintains the state on the Y node in the input scenarios when the pull-up and pull-down network are both inactive.

2.4 Production Rules

Production rule sets (PRS) are compact, abstract representations of a digital CMOS circuit. Each production rule is a construct of the form $G \rightarrow S$, where G is a Boolean expression known as the guard of the production rule, and S is the resulting assignment to a Boolean variable. $x\uparrow$ and $x\downarrow$ are used to represent setting a Boolean variable x to $true$ and $false$

respectively. An example of a production rule that sets z to true when x is true and y is false is: $x \wedge \neg y \rightarrow z\uparrow$.

A production rule $G \rightarrow S$ is called CMOS implementable when either:

1. S is of the form $x\downarrow$ and G consists only of un-complemented variables.

2. S is of the form $x\uparrow$ and G consists only of complemented variables.

CMOS-implementable production rules can be thought of as textual representations of pull-up and pull-down networks in a CMOS circuit. The implementability constraint ensures that the pull-down network consists only of NMOS transistors and the pull-up network consists only of PMOS transistors. The CMOS-implementable PRS of a NAND-gate is:

$$\begin{aligned} a \wedge b &\rightarrow y\downarrow \\ \neg a \vee \neg b &\rightarrow y\uparrow \end{aligned}$$

Note that for combinational logic gates, the guards of the two production rules are complements of each other and specifying either one is sufficient. Hence, for convenience, we use the notation $a \wedge b \Rightarrow y\downarrow$ to represent a combinational gate by specifying only one production rule.

State-holding gates are gates where, as opposed to combinational gates, the pull-up and pull-down network may both be inactive for some combinations of inputs. For these combinations, the gate holds the output corresponding to the previous input. For such gates, e.g. C-elements, both production rules are specified. The CMOS-implementable PRS for a 2-input inverting C-element is:

$$\begin{aligned} a \wedge b &\rightarrow y\downarrow \\ \neg a \wedge \neg b &\rightarrow y\uparrow \end{aligned}$$

For state-holding gates, the complete circuit also consists of a staticizer, apart from the pull-up/pull-down networks. However, we omit the complete PRS description of the stati-

cizer in the notation and it is understood that it exists at every node that is the output of a state-holding gate.

2.5 Synchronization, Slack and Elasticity

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.

The synchronization behavior of a CHP program is a result of the organization of communication actions in the program. Changing the synchronization behavior can modify the underlying computation in a fundamental way. For example, consider the following program that corresponds to a counter.

$$
\begin{aligned}
x := 0; *[[\,|\, & \overline{INC} \longrightarrow x := x + 1; INC? \\
[]\ & \overline{INC2} \longrightarrow x := x + 2; INC2? \\
[]\ & \overline{READ} \longrightarrow READ!x \\
[]\ & \overline{ZERO} \longrightarrow x := 0; ZERO?\ \ |]]
\end{aligned}
$$

Now if we have an environment that executes the following sequence of actions:

$$ZERO!; INC!; INC2!; READ?v$$

then we know that the value of v will be set to 3 in all possible executions. On the other hand, if we change the synchronization behavior of the environment to

$$ZERO!; INC!; (INC2! \parallel READ?v)$$

then the value of v is *non-deterministic*—it could be either 1 or 3. Mapping a CHP program into dataflow components often changes its synchronization behavior. In the example above, since $INC2$ and $READ$ are different channels without any *local* data-dependency,

dataflow decomposition will by default permit their parallel execution [16, 17]—resulting in erroneous executions.

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 [18].

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 [18]; 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 semi-colons 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.

2.6 Static Token Form

Static Token Form (STF) [19] is a particular form of CHP that satisfies some constraints that make it easier to perform program analysis. STF is closely related to other forms such

as Static Single Information (SSI) and Static Single Assignment (SSA) used in traditional compiler literature.

The process of converting into STF consists of several transformations 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 [20]. 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:

$$*[C?c, X?x;$$
$$\quad [c = 0 \longrightarrow Y!x [] c = 1 \longrightarrow Z!x [] else \longrightarrow skip] \ ]$$

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:

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

In case the variable is unused in a branch, a dummy bottom/null variable ($\perp$) 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:

$$*[C?c; [c \longrightarrow Y?x_1 [] \neg c \longrightarrow Z?x_2]; \\ x := \phi(x_1, x_2); X!x]$$

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:

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

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:

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

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.

2.7 Repetitive Event-Rule Systems

Timing analysis of asynchronous logic is more complicated than the equivalent problem in synchronous logic. This is due to the fact that asynchronous logic cannot be readily partitioned into acyclic regions of combinational logic. Many common asynchronous logic gates are inherently state-holding, and the timing behavior of these must be modeled by examining cyclic paths of timing dependencies.

Event-Rule (ER) systems [21] are models used to analyze the timing properties of asynchronous systems. In these approaches, the circuit is abstracted away as a collection of events (abstraction of a signal transition), and the circuit topology is used to construct a graph that captures inter-event dependencies and their timing relations. ER systems model and-causality (also known as max-causality), where an event can occur only after all its predecessors have occurred.

It is known that an RER system with and-causality and fixed delays always exhibits exact periodicity and can be exactly characterized by a periodic function after a finite number of occurrences of each transition [22].

An ER system is a pair E, R where E is a set of events, and $R \subseteq E \times E \times R_{\geq 0}$ is a set of rules that define timing constraints. A rule $r = (e1, e2, \alpha)$ is written as $e_1 \overset{\alpha}{\mapsto} e_2$, and indicates that the event e_2 cannot occur until α time units after e_1 occurs. e_1 is said to be the source of r [denoted $src(r)$], and e_2 is said to be the target of r [denoted $tgt(r)$]. There could be events f for which there are no rules that constrain their timing. In other words, there is no $r \in R$ such that $tgt(r) = f$. These are referred to as initial events.

For a non-terminating computation, the set of events E as well as the set of rules R are infinite. Often systems with non-terminating computations of interest, such as a closed asynchronous circuit, are repetitive. Hence, although the sets E and R are infinite, they can be represented as unfoldings of a compact, finite representation, just like a finite while-

loop can be unrolled to construct an infinite computation. This compact representation is precisely the RER system.

An RER system is a pair $\langle E', R' \rangle$ where E' is a set of transitions, and $R \subseteq E \times E \times R_{\geq 0} \times \mathbb{N}$ is a set of rule templates. The ER system corresponding to $\langle E', R' \rangle$ is given by $\langle E, R \rangle$, where $E = E \times \mathbb{N}$ and R consists of rules of the form $\langle u, i \rangle \overset{\alpha}{\mapsto} \langle \nu, i + \epsilon \rangle$ where $r = (u, v, \alpha, \epsilon) \in R'$ and $i \in \mathbb{N}$. Here, the non-negative integer i is called the occurrence index and ϵ is called the occurrence index offset of r. In a circuit, the occurrence index typically corresponds to the iteration of the circuit on which a particular event happens.

It is sometimes convenient to visualize ER systems as well as RER systems in graphical format. For event rule systems, the acyclic constraint graph is used to visualize timing constraints. The graph is constructed from the ER system as follows: the set of vertices is the set of events E and a rule $e \overset{\alpha}{\mapsto} f \in R$ is displayed as an edge from e to f with an edge weight label corresponding to the delay α. For RER systems, a similar visualization approach is used and is referred to as the collapsed constraint graph. The set of vertices of the graph is the set of transitions E, and each rule template (u, v, α, ϵ) is displayed as an edge from u to v with an edge weight α. In addition, if $\epsilon > 0$, the edges are marked with lines (called "ticks") where the number of lines is the value of ϵ.

In order to see how an RER system captures a repetitive computation, consider the simple example of a 3-inverter loop:

$a \Rightarrow b\downarrow$

$b \Rightarrow c\downarrow$

$c \Rightarrow a\downarrow$

This closed system consists of an infinite repetition of the events (transitions):

$\{a\uparrow, b\downarrow, c\uparrow, a\downarrow, b\uparrow, c\downarrow, ...\}$. Despite being an infinite repetition, we can represent it with a finite collapsed constraint graph, shown in Fig. 2.2. The edges are marked with the delays of the inverters d_{Gi}. We also refer to the collapsed constraint graph as the timing graph throughout this work. A cycle in the timing graph is a set of directed edges that form

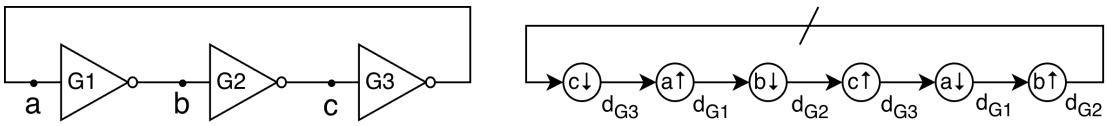


Figure 2.2: A simple ring of inverters and the corresponding timing graph. A tick exists on the edge from $b\uparrow$ to $c\downarrow$.

a closed loop. In this system, the only cycle that exists contains a tick. In general, it is true for every well-formed timing graph (i.e. one that corresponds to a circuit that actually oscillates) that every cycle contains at least one tick.

The cycle mean of a given cycle C is the ratio of the sum of weights (delays) on the edges on the cycle to the sum of the number of ticks on the edges:

$$p(C) = \frac{\sum_{e_i \in C} d(e_i)}{\sum_{e_i \in C} \epsilon(e_i)} \tag{2.1}$$

The critical cycle (C^*) is the cycle with the largest cycle mean and its cycle mean is the critical cycle mean (p^*):

$$p^* = max\ p(C), \quad C^* = argmax\ p(C) \tag{2.2}$$

It is known that an RER system with fixed weights (delays) and max-causality among events is exactly periodic. The principal result to keep in mind is that the performance (throughput) of a closed asynchronous system is determined by the cycle mean of its critical cycle, p^*. Details about the derivation of this result can be found in [22]. Thus, an optimizer that tries to maximize the throughput of an asynchronous system can equivalently minimize the cycle mean of the critical cycle.

Chapter 3

Maelstrom: A Logic Synthesis Technique

Maelstrom is a new synthesis method and corresponding open-source tool that synthesizes CHP programs into asynchronous circuits. The method is agnostic to circuit family, and produces circuits that show significant improvements over the state-of-the-art synthesis techniques for asynchronous circuits in terms of energy, delay and area. The method also supports different datapath implementations and communication protocols. Pre-layout SPICE simulations of generated netlists of several CHP programs in a 65nm node indicate significant performance benefits over the current state of the art. Maelstrom has also been used to successfully synthesize and fabricate a chip from an abstract high-level functional description.

This represents a qualitative improvement in logic synthesis of asynchronous logic from behavioral descriptions.

3.1 Introduction

Logic synthesis is the process of converting an abstract behavioral description of a circuit into a gate-level description that implements the specified behavior. For simple descriptions, there are several "textbook" methods such as Karnaugh maps and Quine-McCluskey minimization that start with a sum-of-products expression for Boolean functions, and then

systematically optimize the Boolean formula. However, as systems become larger, there is a need for the maintenance of internal state of the circuit to support more complex computations. This leads to a separation of the problem into two components: the design of the computation or datapath logic, which is typically combinational logic for arithmetic/logical operations; and the design of the state machine or control logic, which is typically sequential logic that controls the computation performed.

In synchronous design, register-transfer level (RTL) hardware description languages such as Verilog are used to specify the underlying computation in a more user-friendly format. The synthesis methodology translates the RTL into combinational logic and clocked elements using a standard clocking discipline. The problem of synthesizing the state machine (the control path) is quite straightforward in synchronous design since it is entirely controlled by a global timing reference—the clock signal—that is used to perform a state update for the controller. There are several ways to implement state machines, such as using edge-triggered flip-flops, or level-sensitive latches. Further, the communication between several state machines is realized either through implicit communication at each clock edge, or through data transfer protocols such as the classic ready-valid protocol, which uses two additional signals (ready and valid) to perform a data transfer with a shared clock between the sender and receiver.

In asynchronous circuits, there is no global timing reference, and hence the control path must be self-timed in some way. There have been several proposed solutions to this over the decades. One of the first methods to synthesize asynchronous state-machines used Huffman flow-tables [23]. Huffman flow-tables can be compiled into circuits by extending Karnaugh-map style logic synthesis, and by introducing delay lines to form a feedback loop to hold state. More recent methods include burst-mode design [24], which is closely related to synchronous design by virtue of its use of a local timing signal that served the role of the clock. Both methods assume a bound on the delay of local logic, and hence require timing assumptions to ensure correct operation.

At the other extreme are delay-insensitive (DI) circuits, which assume unbounded delays on gates and wires. However, this restricts the class of implementable functions severely [25]; instead, a slightly relaxed version known as quasi-delay-insensitive (QDI) circuits are often used. In the QDI model, wire delays are assumed to be small relative to a sequence of gate delays from an adversarial firing chain of gates, but gate delays can be unbounded. Synthesis approaches for QDI circuits include Martin's synthesis method [26] of handshaking expressions (HSE) to circuits, and Petrify [27], which translates signal transition graphs to circuits. However, these methods require an input description of the system that is at the level of individual signal transitions—much lower level than the RTL-level description used by synchronous circuits. Also, the synthesis algorithms involve complete state-space exploration—a slow step. These two issues render these methods both time-consuming and error-prone, since hand-translation and optimization of the signal transitions is required to obtain high quality circuits [28, 29].

In order to specify a complex asynchronous system at a higher level, we use a popular behavioral abstraction of asynchronous circuits known as Communicating Hardware Processes (CHP). A key requirement for general-purpose logic synthesis of CHP to circuits is preserving the synchronization behavior of the CHP input. This is a more complex problem in the context of asynchronous circuits, since there is no single timing reference that all processes agree upon. Furthermore, changes in the synchronization behavior can in turn introduce new behaviors in the program that did not exist earlier—leading to functional errors [30].

One such synchronization-preserving synthesis tool is the open-source `chp2prs`[31], which is a direct (and unoptimized) implementation of the syntax-directed translation (SDT) approach to logic synthesis for asynchronous circuits [32, 33]. Balsa [9] is an optimized implementation of SDT, providing an implementation that is supposed to be as efficient as a (no longer available) commercial SDT tool called Haste/TiDE [34]. Other approaches to the translation of CHP programs into asynchronous circuits include high-

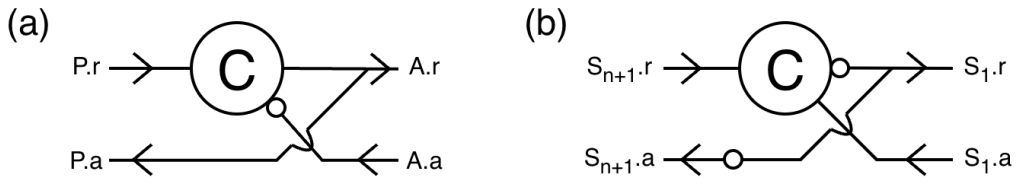


Figure 3.1: (a) C-element buffer circuit. P and A are two channels. The C-element is on the passive side of P (drives the acknowledge) and on the active side of A (drives the request). (b) The initial token buffer, used as an implementation of $*[S_1; S_{n+1}]$, where S_1 is active and S_{n+1} is passive.

level synthesis techniques [17, 35–37], but these approaches modify the synchronization behavior of the input CHP. Hence, they can only be used in limited scenarios where the original program is slack elastic [30].

3.2 Sequential Synthesis of CHP

In this section, we present a new general-purpose approach for translating CHP programs into gate-level asynchronous logic, and an open-source implementation of our technique. Our goal is to perform sequential synthesis, i.e. to obtain a circuit that implements the program *exactly as written*, thereby respecting CHP synchronization semantics. This provides an alternative to syntax-directed translation for general-purpose logic synthesis of asynchronous circuits.

Fig. 3.1 (a) shows the C-element half-buffer circuit. The bubble on one input is an inversion, and can be thought of as having an inverter on that input. In effect, it waits for $P.r \wedge \neg A.a$ to be true (false) before raising (lowering) its output. The exact sequence is:

$$[P.r \wedge \neg A.a]; P.a\uparrow, A.r\uparrow; [\neg P.r \wedge A.a]; P.a\downarrow, A.r\downarrow$$

Upon inspection, it is evident that the C-element really *sequences* two handshakes, a passive one on P and an active one on A. Effectively, it is an implementation of the process whose CHP description is: $*[P; A]$, provided the process has the passive side of channel

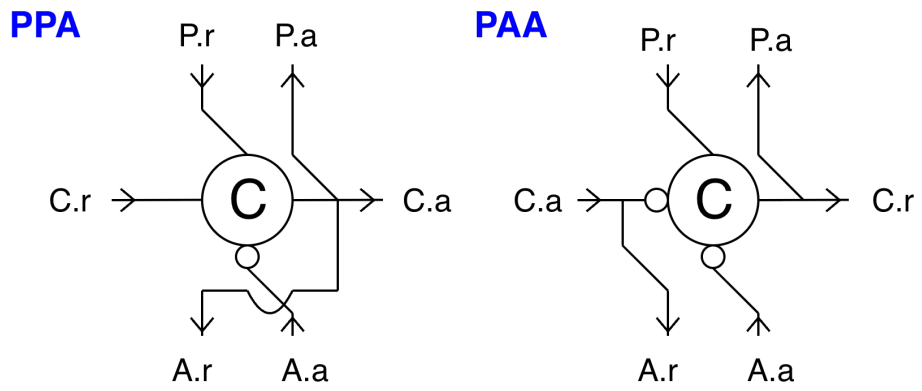


Figure 3.2: The two kinds of sequencer elements, depending on whether the middle channel is active or passive. If active, then the sequencer is the PAA element, and vice versa.

P, and active side of channel A. $P; A$ denotes a channel access on P followed by a channel access on A. The loop $*[...]$ denotes an infinite repetition of the enclosed set of actions. Note that this is a popular circuit, and forms the basis of the control circuits in Sutherland's micropipelines [15] and the first asynchronous microprocessor [38].

The obvious next step from here would be to incrementally extend this to CHP of the form $*[P; C; A]$ where C is another channel, and the process under discussion may have access to either the active or the passive side. In order to perform a three-way handshake sequencing, it might be tempting to use two C-element buffers and connect two of their channels in a way so as to achieve a sequencing on the intervening channel.

However, a more efficient way to perform this sequence of actions can be achieved with a single three-input C-element as shown in Fig. 3.2.

The PAA element (Fig. 3.2) can be directly written as follows:

$$
\begin{aligned}
&*[\ [P.r \wedge \neg A.a \wedge \neg C.a]; P.a\uparrow, C.r\uparrow; \\
&\qquad [\neg P.r \wedge A.a \wedge C.a];\ \ P.a\downarrow, C.r\downarrow\ ] \\
\|\ \ &*[\ [C.a]; A.r\uparrow; [\neg C.a]; A.r\downarrow\ ]
\end{aligned}
$$

where the second process is simply a wire connecting $C.a$ and $A.r$. For the PPA (Fig. 3.2) element, the corresponding process is:

$$*[\ [P.r \wedge \neg A.a \wedge C.r]; P.a\uparrow, C.a\uparrow, A.r\uparrow;$$
$$[\neg P.r \wedge A.a \wedge \neg C.r]; P.a\downarrow, C.a\downarrow, A.r\downarrow\ ]$$

These sequencer elements, which sequence three handshakes, form the fundamental building blocks for program synthesis. As the name suggests, they provide the ability to implement a sequence of actions, which at an abstract level, is precisely the CHP description of a process. These elements, as well as the remaining control circuits described later for each CHP construct, constitute a control circuit library, which is used as a basis set for implementing programs. All circuits in this library, which will be described hereafter, are novel contributions of this work.

3.3 Linear Programs

In order to build up to the complete synthesis methodology for CHP, we start with the simplest kind: linear programs. Linear programs are those where the CHP is of the form $*[C_1; C_2; ...; C_n]$, where each of the terms C_i are elementary actions, i.e. assignments to local variables, sends or receives. In order to synthesize this, note that each semicolon in the program corresponds to a sequencing. Hence, this CHP can be rewritten as:

$$*[S_1; C_1; S_2] \quad \| *[S_2; C_2; S_3] \quad \| ...$$
$$\| \quad *[S_n; C_n; S_{n+1}] \ \| \quad *[S_1; S_{n+1}]$$

where each of the S_i are fresh channels. This rewriting is useful since we already have circuits for the first n programs and synthesis would involve merely connecting them appropriately, under the assumption that the first channel action in any of the new programs is passive, and the third is active. However, under this assumption, the last program, i.e. $*[S_1; S_{n+1}]$ is not of the form implementable by the sequencer elements, since it is an active action followed by a passive action (enforced due to the other sides being passive and active respectively). In order to implement this particular program, we need a special

initializer circuit known as an initial token buffer (ITB), shown in Fig. 3.1 (b). As before, the bubbles represent inversions, and since there is an inversion on the output of the C-element, the initial state for the output is high. From this, it is easy to see that the sequence of transitions for this circuit is:

$$
\begin{aligned}
*[\ \ & S_1.r\uparrow, S_{n+1}.a\downarrow; [S_1.a \wedge S_{n+1}.r]; \\
& S_1.r\downarrow, S_{n+1}.a\uparrow; [\neg S_1.a \wedge \neg S_{n+1}.r]\ \ ]
\end{aligned}
$$

which corresponds to performing an active handshake on S_1 and a passive handshake on S_{n+1}, in parallel. In other words, this circuit implements $*[S_1, S_{n+1}]$, which is slightly different from what was required, which was $*[S_1; S_{n+1}]$. However, this is not an issue since S_1 strictly preceding S_{n+1} is enforced by the rest of the circuit.

In order to understand the circuit topology, notice that the rewritten CHP forms a ring, with each program synchronized with two other programs, which are one step ahead and one step back. Finally, the ITB 'ties up' the first and last ones, closing the ring. Hence, any linear CHP can be implemented using a ring of C-elements, as we have just demonstrated. The channels, implemented by the request and acknowledge wires, form the interface/connections within the ring.

Maelstrom performs precisely this in order to synthesize a linear program. For each action in the program, a sequencer element is instantiated, depending on the type of the action (active/passive). The channels on either side of these elements are then connected in the correct order, forming a chain. Finally, an ITB is instantiated, connecting the passive (incoming) channel of the first sequencer element and the active (outgoing) channel of the last sequencer element, to close the chain up into a ring.

Finally, we observe that the behavior of the ring of sequencer elements is such that the first half-phase (assertion of request and corresponding acknowledge) of handshakes on all the channels S_i are performed first, followed by the second half-phase (deassertion of request and corresponding acknowledge) on all the channels. Essentially, there is a

wave of 0-to-1 transitions propagating through the circuit, followed by a wave of 1-to-0 transitions.

3.4 Parallel Actions

In prior sections, we have seen how to synthesize elementary actions and linear sequences of elementary actions. However, there are several more constructs in CHP which need to be handled in order to obtain a complete synthesis method. The first of these is the parallel construct. The parallel construct allows several elementary actions to happen in parallel, i.e. in any order, as opposed to the sequence, which defines a single order on them. Consider the CHP fragment: $*[..; B, C; ..]$. Here, B and C may complete in any order. Our current synthesis of linear programs is insufficient to handle this, since we can only place a single sequencer element per action and the connectivity defines the order in which they perform the actions. Hence, we need another element, a parallelizer.

The two-way parallelizer is shown in Fig. 3.3. The $prev$ and $next$ channels connect to the rest of the chain of sequencers. When $prev.r$ is raised, the parallel split activates both sequencer elements. The acknowledge on $prev$ is sent back only when both sequencer elements complete (by the C-element in the parallel split). Further, the portion of the circuit following this parallel is only activated when both actions B and C complete (by the C-element in the parallel merge. Finally, the acknowledge is split out into both sequencer elements. The circuits for the N-way parallelizer would involve having N-input C-elements in the parallel split and merge, instead of 2-input C-elements.

3.5 Datapath Synthesis

In the previous sections, we discussed the synthesis of simple CHP programs, yet this is only the half of the picture. The C-elements that implement each elementary action

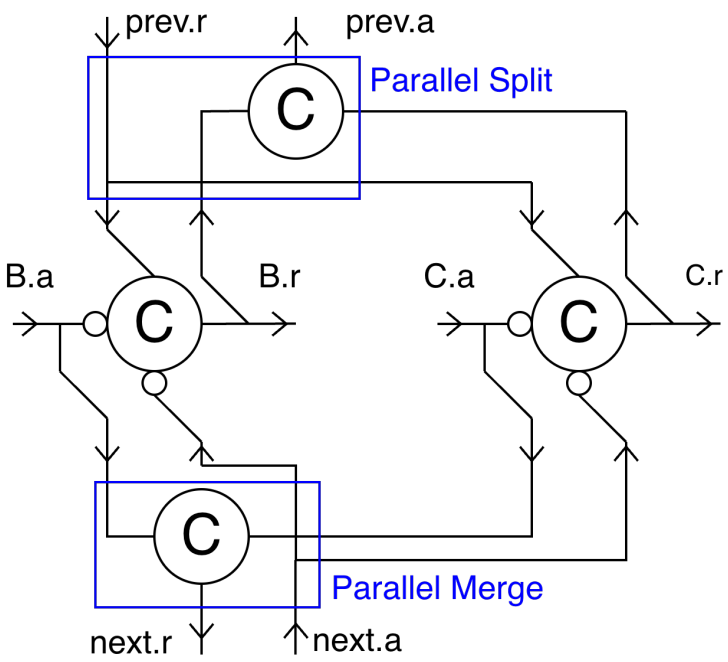


Figure 3.3: The two-way parallelizer element, composed of two parts: the parallel split and the parallel merge.

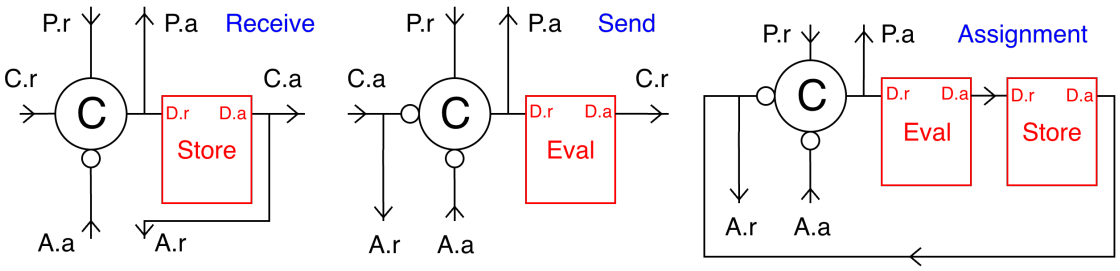


Figure 3.4: Implementations of the elementary actions of CHP using the sequencer elements. Store elements are data storage elements that capture values. Eval elements are combinational circuits that compute some function of variables. Note that an assignment is equivalent to the combination of a send and a receive, from the point of view of the datapath. Hence a send-receive pair is typically known as a "distributed assignment."

also need to communicate with the datapath to actually perform the action. In the case of receives, the value that is received needs to be captured in a state-holding element so that it is accessible to the process later. In the case of sends, the expression to be sent needs to be computed and routed to the data wires of the channel that expose the value, in order for the environment to read them. In the case of assignments, both are required, the computation

of a value and the placement in a data storage element. Fig. 3.4 shows the exact interaction between the control and data components. The channel between the control and datapath, D, acts as a command channel, which causes the datapath component to perform its action.

The actual implementation of these expression computation (Eval) and data storage (Store) elements depends on the choice of datapath circuit family. The conventional case where bits are encoded on a single-rail, is known as the bundled-data datapath family. Alternatively, instead of representing bits as single wires, encoding them in one of several possible 1-of-N codes results in the quasi-delay insensitive (QDI) datapath. Various types of QDI datapaths have been designed previously [39].

For the bundled data circuit family, we use level-sensitive latches as the storage (Store) elements, and use pulse-generators in order to briefly make the latches transparent and capture values. The acknowledge emanating from the Store element is simply a delayed version of the request, with a delay that corresponds to the capture delay of the latch. For the computation (Eval) blocks, we need combinational logic that implements functions of Boolean variables. We use the ABC logic synthesis and verification system to generate these circuits. For the Eval blocks, similar to the Store blocks, the acknowledge returning to the control is a delayed version of the request, with the delay matched to the delay of the combinational logic that implements the necessary computation.

3.5.1 STF-Style Datapath

The choice of datapath family is crucial to the design process, but it is not the only one. Even within a single circuit family, there are several ways of implementing the same datapath when variables are assigned and used several times. For example, consider the following CHP:

$$*[A?x; B!(x+1); C?x; D!(x+2)]$$

Here, the variables x is assigned twice. Hence, it is necessary to allow for the value

of x to be written/updated from two places. When translating this to hardware, x can be implemented with either a single Store element with two write-ports, or as two Store elements. For the bundled datapath family, this would correspond to a single W-bit latch with a W-wide 2-to-1 mux, or two individual W-bit latches (where W is the bitwidth of the variable x). In general, for a variable that is assigned N times in a program, it can be implemented as a single latch with an N-to-1 mux, or as N separate latches. To determine which strategy is better, it is necessary to count the number of transistors that would be required to implement each.

First, notice that the problem of sharing pulse generators (shown in Fig. 3.5) in the single latch implementation is non-trivial. In general, each variable (implemented by a latch) can be written from several different parts of the program, and the pulse generator must trigger when each of these control signals go high, irrespective of the state of the others. Implementing this with a single pulse generator would require a complex state machine that produces a rising edge when each of the control signals are asserted. Due to this, it is evident that the cheapest way to implement this is in fact with several pulse generators - one per control port - with the final latch clock signal being the logical OR of the outputs of these circuits. Hence, we compare our datapath implementation style with this style of using pulsed latches.

Suppose a variable is assigned N times in a program, and a single pulse generator can be implemented with k transistors, where k is a constant that is determined by the required pulse-width for the latch to successfully capture data. We will assume that the variable is a 1-bit variable for simplicity; the results can be easily scaled to get the desired numbers for an arbitrary bitwidth. In the first implementation, where we use a single latch with muxes, we require 12 transistors for the latch, $10(N-1)$ transistors for the N-way mux, kN for the N pulse generators, $3N$ for the N-way OR-gate, which results in the required number

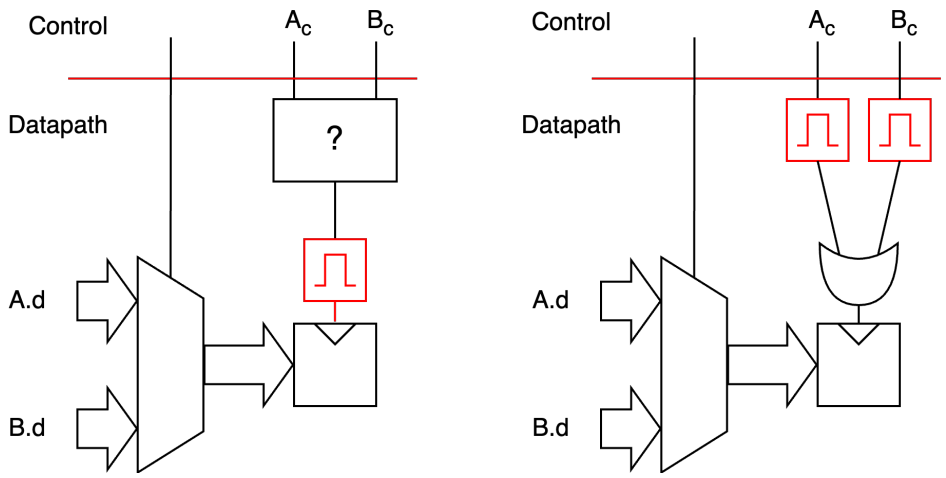


Figure 3.5: The problem of sharing pulse generators across control ports for the same latch. The most efficient way to implement this is to have one pulse generator per control port.

of transistors:

$$N_{FET} = \begin{cases} (13+k)N+2, & \text{if } N > 1 \\ (12+k), & \text{if } N = 1 \end{cases}$$

Note that the $3N$ for an N-way OR-gate is actually a lower bound, and the actual count is larger. In the second implementation, we require $12N$ transistors for the latches, and kN transistors for the pulse generators, resulting in:

$$N_{FET} = (12+k)N$$

which is uniformly better than the first implementation. Hence, we choose this for our datapath implementation. This technique is effectively equivalent to renaming variables every time they are assigned, and replacing accesses of variables between two assignments with accesses of the correct fresh variable. Thus each fresh variable is assigned exactly once, and only accessed until another of the fresh set is assigned. Effectively, we rewrite the CHP as:

$$*[A?x_0; B!(x_0+1); C?x_1; D!(x_1+2)]$$

This is very similar to a standard representation in software compilers, known as static single assignment form, which makes program analysis easier [40]. Here, it has the added benefit of reducing the transistor count.

3.5.2 Initial Conditions and Loop-carried Dependencies

So far, we have assumed that the CHP description of a process is of the form $*[P]$, where P is some sequence of actions. However, not all programs can be described by this template. In some cases, there is a need for a defined initial state, followed by a non-terminating program. For example, consider the accumulator:

$$a := 0; *[X?x; a := a + x; A!a]$$

Here, the initial assignment, $a := 0$ is crucial for the correct operation of the process. Note that a is also a loop-carried dependency, i.e. the value of a on one iteration of the loop is needed for the next iteration. In general, we can have a list of initial conditions, resulting in CHP of the form:

$$x_1 := v_1; ...; x_n := v_n; *[\ P\ ]$$

where $\{x_k : k = 1..n\}$ is some subset of all the variables used in the program. Hence, we need additional circuitry in order to implement this. In the context of our current implementation scheme, this is quite straightforward. For each variable that is initialized, we simply instantiate a latch that is initialized to the required value on reset, and the downstream program refers to the value on this latch until the next assignment to the variable. Further, an additional assignment block that stores the 'last' value of the variable in the aforementioned initial-value latch is created, effectively implementing the loop-carry action that is needed for the next iteration of the loop to proceed correctly. This assignment block occurs last in the sequence of actions in P, after all actions in the original program have been completed, guaranteeing that the correct value to carry around the loop is available.

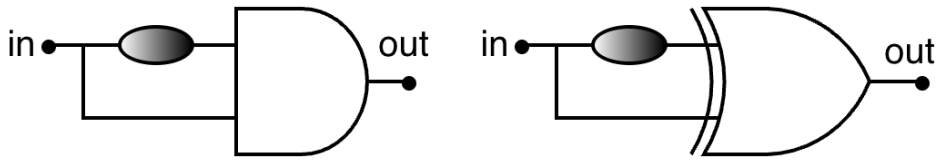


Figure 3.6: Left: The pulse generator for the four-phase datapath. Right: The pulse generator for the two-phase datapath.

3.5.3 Two-Phase vs. Four-Phase Circuits

The circuit synthesis technique that we present here allows the user to choose between two styles. The first is where the datapath is activated/pulsed only on the positive half-phase of the handshake wave propagating through the controller with the negative half-phase being used purely for reset. We refer to this as the four-phase style, as one iteration of the CHP program corresponds to an entire four-phase handshake propagating through the circuit. The other is where the datapath is pulsed on both half-phases. We refer to this as the two-phase style for the same reason as before.

Supporting these two styles require the pulse generators to activate on either one or both control edges. The topologies to satisfy this requirement are shown in Fig. 3.6. Finally, as we will see in the following sections, the circuits for implementing selections will also require some modifications depending on whether we are interested in a four-phase or two-phase synthesis style.

3.6 Deterministic Selections

In this section, we deal with the first construct in CHP that encodes conditional executions. Consider the following CHP fragment:

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

For the deterministic selection, each of the guards are pairwise mutually exclusive, i.e. at most one of the guards may be true. For the program to be free from deadlock,

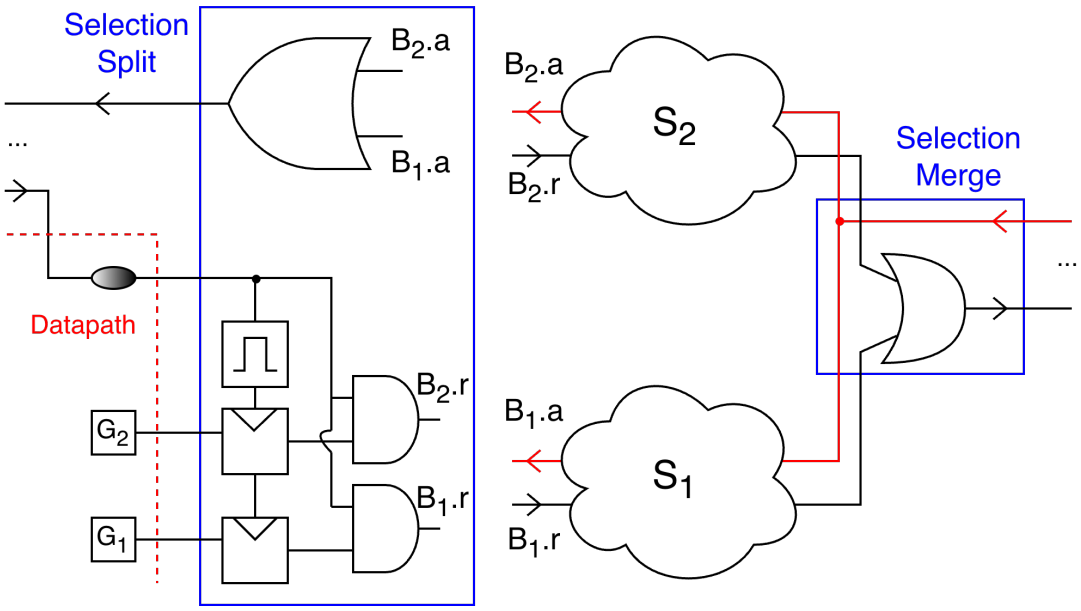


Figure 3.7: Implementation of a two-way selection construct for a four-phase datapath, composed of two parts: the selection split and the selection merge. The interface with the datapath is as shown.

however, it may not be the case that none of the guards are true when the selection guards are evaluated; such CHP can be written but is most likely a user error due to the fact that CHP selections are blocking, i.e. if none of the guards are true, then the program execution does not proceed past it, but instead waits for some guard to become true. Once a guard is determined to be true, the corresponding branch, which can contain any CHP, is executed. After the completion of the branch, the execution proceeds to the portion of the program after the selection.

From the description of selections, it is evident that this is the first construct that requires a causal interaction between the datapath and the control circuits. The sequence of actions/executions in prior programs could be statically determined, but that is not the case here. The choice of branch depends on the values of internal variables and may change from one execution of the top-level loop (`*[...]`) to the next. In order to implement this, we notice that we simply need to split the request out forwards and merge the acknowledge backwards, similar to the parallel case, but to the single correct branch, instead of all

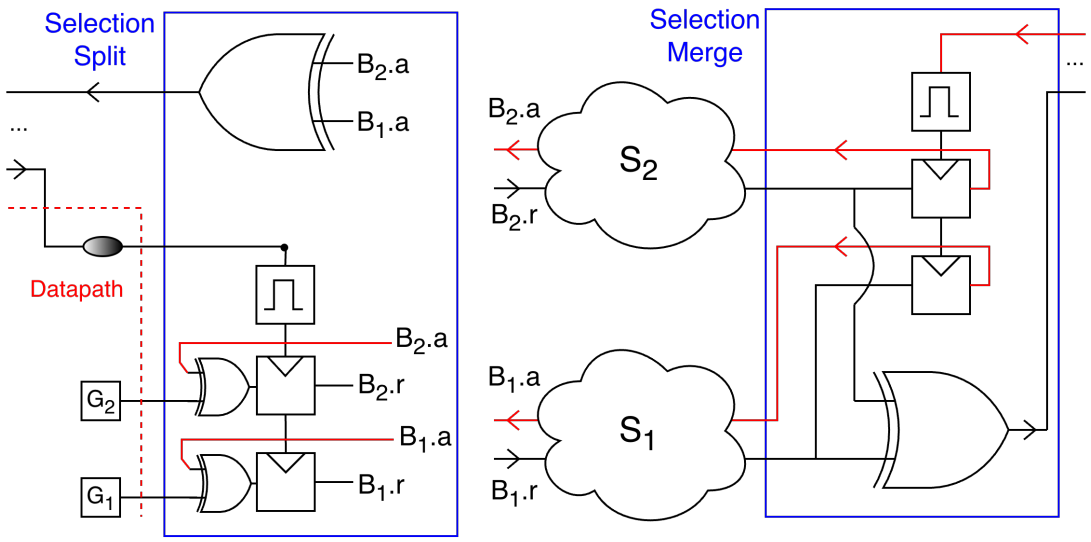


Figure 3.8: Implementation of a two-way selection construct for a two-phase datapath, composed of two parts: the selection split and the selection merge. The interface with the datapath is as shown.

branches.

The implementation of a two-way selection for a four-phase datapath is shown in Fig. 3.7. The incoming request is routed to the datapath in order to read out (Eval) the values of the guards. For simplicity of exposition, they are shown to be encoded on a single rail, i.e. the bundled data encoding, but other encodings are also possible, as discussed earlier. This guard evaluation can be thought of as a multiple-output Eval block, with the acknowledge delay being the largest of the delays of the evaluators for each of the guards. The read acknowledge, which implies that the guard values have settled, is used by the selection split to activate the appropriate branch. The selection split also propagates the acknowledge from the activated branch backwards. As before, the interface with the sub-programs are in terms of request-acknowledge channels. Once the activated sub-program completes, the control passes downstream to the rest of the program, via the selection merge that combines the requests and splits the acknowledges correctly. As in the case of the parallel construct, this too can be easily extended from a two-way to an N-way construct.

The two-way selection circuit for a two-phase datapath is shown in Fig. 3.8. In the four-phase circuit, one can imagine a rising edge traveling through the branch in the selection controller in the positive half-phase followed by a reset half-phase propagating through the *same* branch before any other branch is used. However, the circuits that implement selections in two-phase are more complex than in the four-phase case as they need to, in some sense, 'remember' each branch that was taken and route the control pulse propagating through the selection on a particular iteration. The four-phase circuit has the luxury of an entire dedicated phase just for reset and thus requires a simpler circuit for implementing selections. In order to see this more clearly, consider this simple CHP for a merge:

$$*[C?c; [c = 0 \longrightarrow X?x [] c = 1 \longrightarrow Y?x]; Z!x]$$

Suppose that the sequence of values on the channel C is $0, 1, 0$. Now, the 4-phase circuit simply activates the $c = 0$ branch, resets, activates the $c = 1$ branch, resets, activates the $c = 0$ branch, and resets. Hence, we know that the positive-, or data- half-phase is always a rising edge that propagates through the circuit. In the 2-phase circuit for the same program, the circuit must send an active-high control wave through the $c = 0$ branch, do the same to the $c = 1$ branch without affecting the $c = 0$ branch, then send an active-low control wave through the $c = 0$ branch without affecting the $c = 1$ branch. At the end, the circuit is in a different state than when we started, with the control elements in the $c = 1$ branch still holding a high. This orchestration that is implemented with XOR-gates and one-bit latches adds overhead.

3.6.1 Φ-functions and Merging Logic

Our datapath synthesis was designed with simplicity, and therefore transistor count, in mind. However, selections and conditional executions introduce an inconvenience. To illustrate the problem, we consider the following CHP:

$$*[C?c; [c = 0 \longrightarrow x := 4 [] c = 1 \longrightarrow x := 8]; X!x^2]$$

If each x was renamed every time it was assigned, then it is not possible to determine before runtime the correct latch whose output must be used to calculate the value that is sent out over the channel X. In order to handle this, we introduce logic at the end of every selection, that essentially acts as a value merge, selecting the correct latch based on which branch was taken in that selection. We only perform this for variables that are used after the selection, in the downstream program.

In the static single assignment form mentioned earlier, this is known as a Φ-function, which picks the correct value based on which branch was taken in a program. This analysis of detecting uses of variables is part of a larger live variable analysis pass that is performed as a pre-processing step. The instantiation of merging logic is performed automatically as part of Maelstrom's synthesis flow.

3.7 Non-Deterministic Selections

The synthesis of non-deterministic selections is similar to the deterministic selection in the sense that there is still a splitting of the control into one of several branches and a merging at the end, with MUXes to combine values of variables appropriately. However, non-deterministic selections may contain guard expressions where probes appear, and may have multiple guards true at evaluation, necessitating some added complexity.

The circuit makes use of a killable arbiter [41] in order to safely implement a non-deterministic selection (NDS) that is compatible with the rest of Maelstrom's circuits. Fig. 3.9 shows the two-way non-deterministic selection circuit for a 4-phase datapath. The $ctrl$ port connects the circuit to the preceding part of the ring. The B_1 and B_2 ports connect to the two branches of the selection. The execution of the circuit is as follows: the asymmetric C-elements C_1 evaluate the guards when triggered by the $ctrl$ port. As both

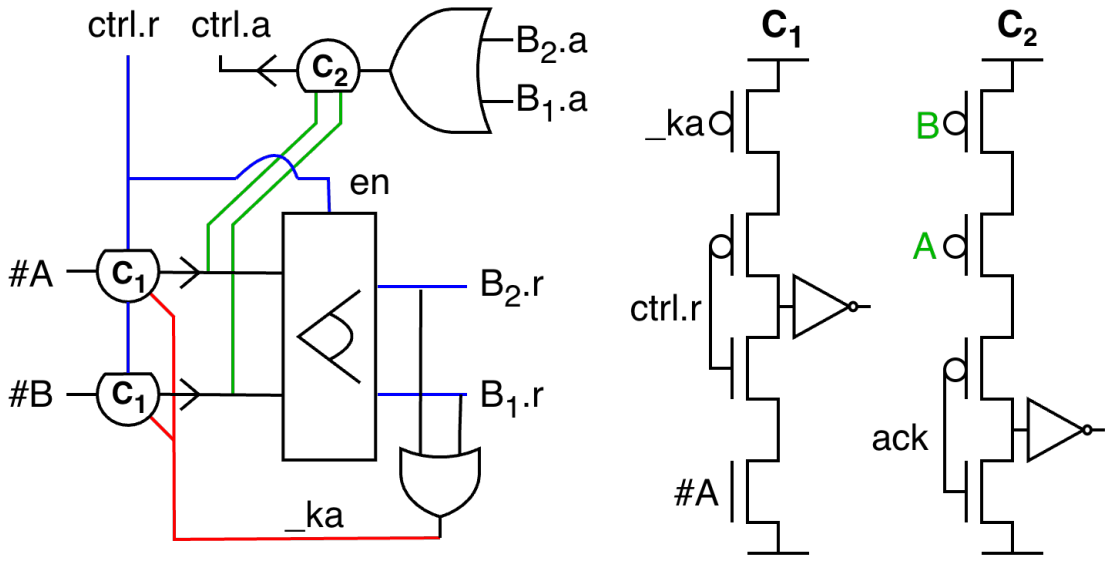


Figure 3.9: The killable arbiter-based implementation of a 2-way non deterministic selection for the 4-phase datapath.

guards can simultaneously be true, we use a killable arbiter to pick one of the branches. The $ctrl$ port also activates the arbiter by raising the enable signal. The chosen branch then executes. Upon reset, the $ctrl$ port disables the arbiter which causes both its outputs to go low and trigger reset in the branches. These transitions are acknowledged via the OR-gate that generates the active-low kill-acknowledge (k_a) signal. The C_1 elements then set their output low. These two transitions are also acknowledged by the C_2 element which then propagates the lowering of $ctrl.a$ backwards. The necessity for complexity in this circuit arises from the fact that the guards may or may not become false once a branch is activated as a branch may contain a communication action on a channel that is probed, and the environment may change the state of the probe at any time.

For the 2-phase datapath case (circuit shown in Fig. 3.10), there is more synchronization necessary as we do not have the luxury of a full, dedicated reset phase. The operation is as follows: Initially all gate outputs are low. The assertion of $ctrl.r$ enables the arbiter, which makes a decision. Upon the assertion of k_a, the latch blocking $ctrl.r$ opens and the latches blocking $B_i.r$ close. The delay line ensures that these latches close before the ris-

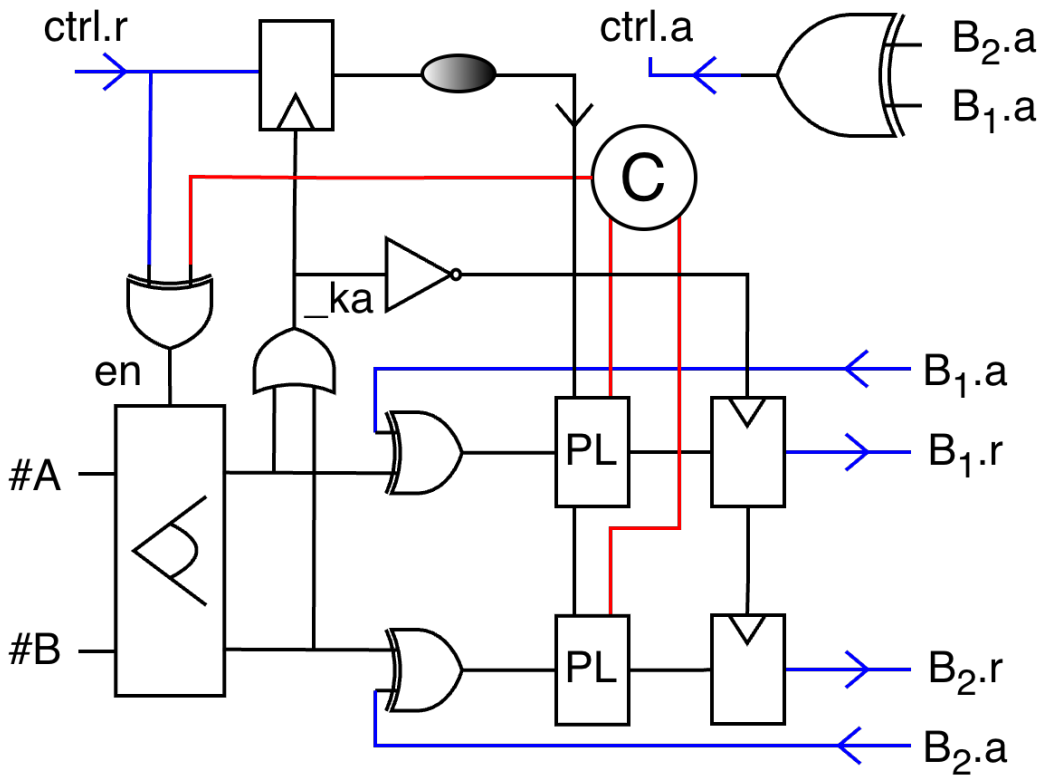


Figure 3.10: The killable arbiter-based implementation of a 2-way non deterministic selection for the 2-phase datapath.

ing edge can reach the PL blocks (these are pulsed latches, which comprise of a latch and a pulse generator which creates a pulse on both the rising and falling edges of the input). The pulsing of the PL blocks latches the arbiter's decision and then disables the arbiter, via the XOR-gate feeding back to the en signal. Once the arbiter has been disabled, the latches open, activating the appropriate branch. On the next data-phase, which happens when $ctrl.r$ de-asserts, the reader may confirm that the same sequence of events repeats.

The general N-way NDS is built upon these two-way selection circuits. The circuit comprises of $N-1$ two-way NDS circuits with the second branch of one feeding into the control of the next one, effectively starting the evaluation of the next guard if the currently evaluated one was false. This effectively implements a sequential evaluation of the guards.

Note that there is an issue when dealing with negated probes in selections: it is possible

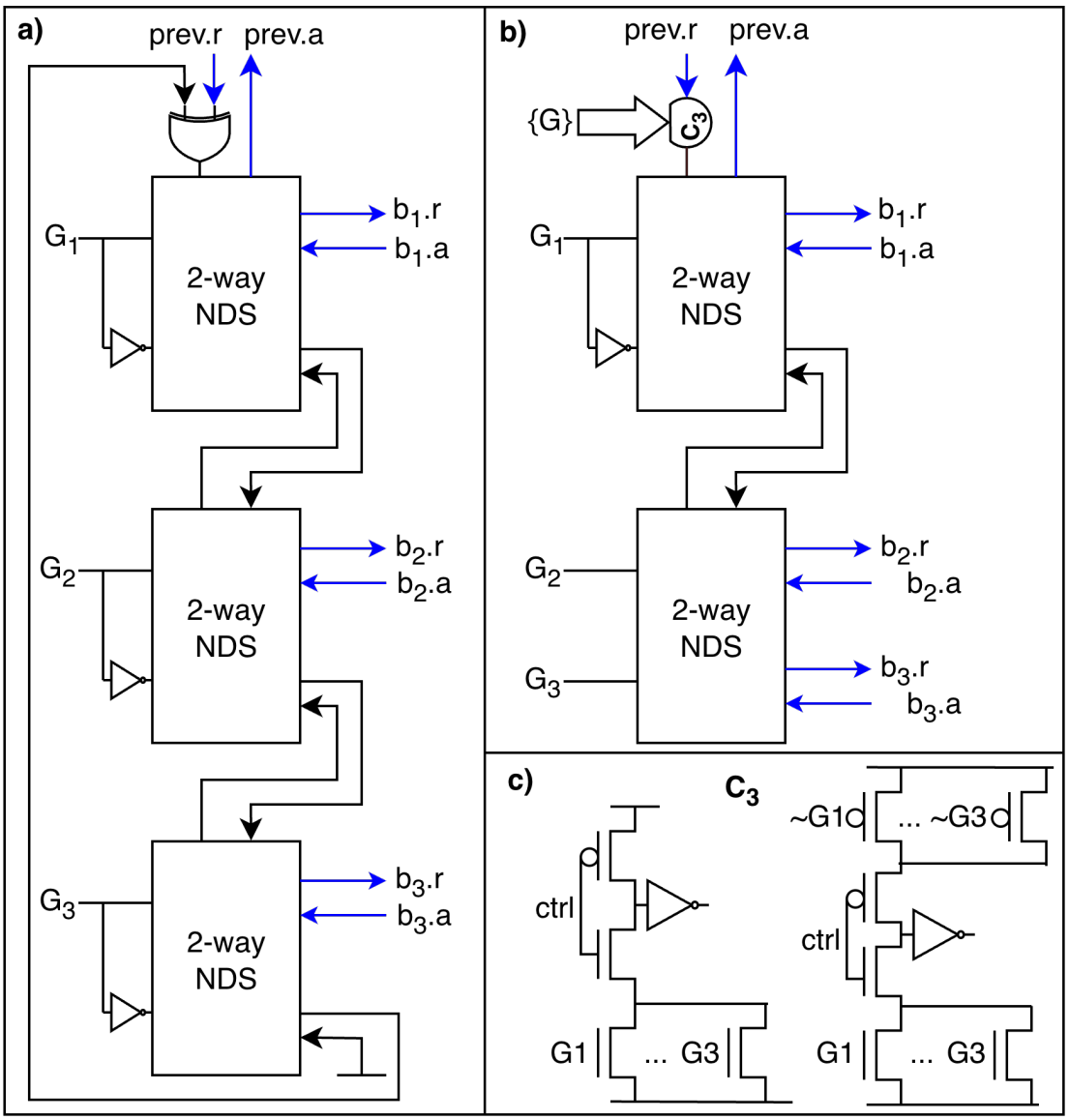


Figure 3.11: (a) Implementation of a general N-way non-deterministic selection where probes and negated probes may be present. (b) Implementation of an N-way non-deterministic selection when probes occur, but only in the positive sense. (c) Implementation of the $C3$ gate, for the 4-phase (left) and 2-phase (right) datapaths.

that none of the guards are true when the circuit begins its evaluation. Further, it possible that a guard that was evaluated and deemed to be false can become true after the evaluation proceeded past it. This is only an issue in sequential evaluation of guards. The problem does not exist if concurrent evaluation of guards is performed, but the circuit complexity

of this is much higher than the sequential case. Fig. 3.11(a) shows the general N-way NDS. The circuit implements a round-robin evaluation of the guards and picks the first true guard. If all the guards are false, it resets and re-evaluates them all in order again.

For the case of stable guards, i.e. there are no negated probes in any of the guard expressions, the above circuit can still be used but it is wasteful in terms of energy if all guards are false for a long time, due to multiple re-evaluations. In order to improve on this, we exploit the fact that with stable guards, once a guard is true, it cannot become false until the circuit performs an action that is observable to the environment. Hence, we can delay the evaluation until we know that at least one guard is true. The circuit in Fig. 3.11(b) contains a pre-checking circuit element ($C3$) that does precisely this, and fires when the control and at least one of the guards is true. This method also has the added benefit of reducing the number of arbiters required by one. Fig. 3.11(c) shows the exact implementation of $C3$ for the four-phase and two-phase datapaths.

Finally, consider the case of the single-branch selection: $[G \rightarrow S]$. This is an unnecessary construct if the guard G only consists of local variables, as we assume that there are no shared variables across processes. A process that contains this would either deadlock or always find G to be true on evaluation. Only the latter case is relevant and here the snippet can be replaced simply with S. However, if G contains probes, then it becomes a useful wait condition and we can implement this with no arbiters at all. Following the same notation as in Fig. 3.11(a,b), the production rule set (which can be trivially made CMOS-implementable) is simply:

$$
\begin{aligned}
prev.r \wedge G &\rightarrow b1.r\uparrow \\
\neg prev.r &\rightarrow b1.r\downarrow \\
b1.a &\rightarrow prev.a\uparrow
\end{aligned}
$$

This construct, combined with latch elimination allows us to write essentially 'latchless' programs that are purely control circuits in CHP. For example, consider a buffer

that tightly synchronizes the actions on the two channels: $*[L?x \bullet R!x]$.[1] In order to obtain a circuit that achieves this behavior via synthesis, one can write: $*[[\overline{L} \rightarrow x := L]; R!x; L?]$, or better yet: $*[[\overline{L} \rightarrow R!L]; L?]$ (which makes use of channel expressions). These two programs are semantically equivalent to the first one as in all three cases, the environment must perform $L! \parallel R?$ in order for the system to not deadlock.

The circuit implementation for these programs results in zero latches/storage elements and consists purely of control C-elements. This structure is even more effective when creating latch-less (and slack-less) copies, splits etc., especially when the bitwidth of the data variable grows and one wants to avoid the overhead of latching it (or avoid introducing slack). For example, a latch-less 3-way copy process can be written as: $*[[\overline{L} \rightarrow R_0!L, R_1!L, R_2!L]; L?]$.

3.8 Loops

The next target CHP construct is the repetition, also known as a loop. So far, we have implicitly used a special case of a loop to represent the top-level infinite repetition that characterizes a circuit. In general, a CHP loop is of the form:

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

The guards are Boolean-valued expressions that are required to be pairwise mutually exclusive, but may not be an `else`. As opposed to selections, loops are non-blocking, i.e. if none of the guards are true, then control passes past the loop to the downstream program. If the loop consists of a single branch whose guard is the elementary Boolean expression `true`, then the loop reduces to an infinite loop: $*[true \rightarrow S]$. The top-level infinite repetition $*[S]$ that has been implicitly used so far is essentially a shorthand notation for $*[true \rightarrow S]$. Conceptually, loops are the most complex constructs in CHP and the

[1] In Martin's CHP, the bullet operator is the tightest form of synchronization, ensuring that the number of completed actions on the two channels are always the same.

problem of synthesizing a circuit that implements them in the general case is a difficult one. In the simple case of a single branch, with a guard that is identically true, we have shown in prior sections that a ring of C-elements is a valid implementation. However, extending it to several branches directly is quite complex and results in a circuit that is expensive. Firstly, note that top-level loops with multiple branches, i.e. ones of the form:

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

can be rewritten into a single-branch loop with a multi-branch selection as:

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

Note that these are equivalent since loop guards can only consist of local variables. If the first program is non-terminating, then the behavior of the second program is indistinguishable from that of the first. If the first program terminates, i.e. all guards are false, then the second program would be deadlocked since selections in CHP are blocking. In this case as well, the behaviors of the two programs are indistinguishable from the perspective of an external observer. Hence, this rewrite is justified.

This addresses the case for when the multi-branch loop is at the top-level. Once this rewrite has been performed, any remaining multi-branch loops must be within another loop. In general, these are of the form described at the start of this section. Instead of synthesizing these loops directly, we excise them into a separate process. This is justified by the fact that the circuit complexity for the two strategies is almost identical.

In order to derive the 'excision' strategy for these loops, recall that we have already computed live variable information at all points in the program. Hence, we can extract this loop out into a separate process, and insert data communications appropriately in order for the loop to perform the correct computation. As a simple example, consider the following CHP, which computes the greatest common divisor of two numbers via Euclid's algorithm:

$$*[\ X?x; Y?y; *[x > y \longrightarrow x := x - y$$
$$[] y > x \longrightarrow y := y - x]; O!x\ ]$$

Here, the internal loop requires two values/inputs to operate (x and y) and returns another value/output (x). In general, the number of input and output variables can both be greater than one. In this case, we simply concatenate them all and transmit them as one, in order to minimize the umber of channel actions required, which directly minimizes the control overhead.

To perform the excision, we factor out the loop into a separate infinite loop, similar to factoring out code into a function call. The rewritten CHP results in two loops operating in parallel, properly synchronized through the fresh channels L_s and L_f:

$$
\begin{array}{l}
*[\ X?x;\ \ Y?y;\ \ L_s!\{x,y\}, L_f?x_2;\ \ O!x_2\ ]\ \ \| \\
c := 0; \\
*[\ [c = 0 \longrightarrow\ \ L_s?\{x_1, y_1\}, c := 1 [] else \longrightarrow\ \ skip]; \\
\quad [x_1 > y_1\ \ \longrightarrow\ \ x_1\ :=\ \ x_1 - y_1 \\
\quad [] y_1 > x_1\ \ \longrightarrow\ \ y_1\ :=\ \ y_1 - x_1 \\
\quad [] else\ \ \ \ \longrightarrow\ \ L_f!x_1, c := 0]\ ]
\end{array}
$$

The reader can verify that the CHP is equivalent to the original and is now composed of two programs operating in parallel, each of which contains only the top-level simple loop, and can hence be synthesized based on techniques discussed in prior sections.

We now state the general method to excise loops. Consider any nested/internal loops of the form below:

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

The appropriate rewritten CHP for this is:

$$*[..; L_s!\{x_1, x_2, .., x_n\}, L_f?\{y_1, y_2, .., y_m\}; ..] \quad \|$$
$$c := 0;$$
$$*[\ [c = 0 \longrightarrow L_s?\{x'_1, x'_2, .., x'_n\}, c := 1$$
$$\quad [] c = 1 \longrightarrow skip];$$
$$\quad [G'_1 \longrightarrow S'_1 [] G'_2 \longrightarrow S'_2 .. [] G'_n \longrightarrow S'_n$$
$$\quad [] else \longrightarrow L_f!\{y'_1, y'_2, .., y'_m\}, c := 0]\ ]$$

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 variable c is a one-bit variable and hence one of the two branches in the first selection will always be true. This transformation is performed automatically as a pre-processing step on the input CHP, prior to circuit generation. Finally, note that there can be several levels of nesting of loops within other loops and in this case, the excision is performed bottom up, recursively.

3.8.1 Do-Loops

The method for handling do-loops is quite similar to that of loops. Consider any internal do-loop of the form $*[..; *[S_1 \longleftarrow G_1]; ..]$. The appropriate rewritten CHP for this is:

$$*[..; L_s!\{x_1, x_2, .., x_n\}, L_f?\{y_1, y_2, .., y_m\}; ..] \quad \|$$
$$c := 0; *[[c = 0 \longrightarrow L_s?\{x'_1, x'_2, .., x'_n\}, c := 1$$
$$\quad [] c = 1 \longrightarrow skip];\ \ S'_1;$$
$$\quad [G'_1 \longrightarrow skip$$
$$\quad [] else \longrightarrow L_f!\{y'_1, y'_2, .., y'_m\}, c := 0]]$$

where G'_1 and S'_1 have the same meaning as in the case of loops. As before, we are left solely with CHP contructs for which the synthesis procedure is already defined. Finally, an interesting observation to be made is that loops can be converted to do-loops and vice versa. The conversion from a do-loop to a loop is straightforward. The conversion of a

loop to a do-loop can be performed as follows. Consider a loop of the form: $*[G_1 \rightarrow S_1 [] G_2 \rightarrow S_2 .. [] G_n \rightarrow S_n]$.

This can be rewritten into a do-loop, with a simple guard that essentially checks if the original loop has terminated, and a selection inside it as shown below:

$$*[\ \ c := 1; [G_1 \longrightarrow S_1 [] G_2 \longrightarrow S_2 .. [] G_n \longrightarrow S_n$$
$$[] else \longrightarrow c := 0] \ \ \longleftarrow \ \ c = 1\ \ ]$$

This conversion to do-loops is also a part of the pre-processing that is performed in Maelstrom. Do-loops also have the property of being easier to deal with from the perspective of data-dependency analyses, since the body of the loop is guaranteed to execute at least once. Since loops and do-loops are inter-convertible as shown above, in the next section, we will assume that loops have already been converted to the equivalent do-loops and deal solely with these.

3.9 Multiple Channel Accesses

The synthesis procedure described so far has overlooked an important issue, which will be addressed in this section. In order to understand the source of the problem, first consider the following simple CHP which contains a multiple send over the channel X: $*[X!0;\ Y!1;\ X!2]$

This is a simple linear program of the form described in Section 3.3, and at first glance, might look like it can be synthesized directly. According to the aforementioned method, this would be rewritten as:

$$*[S_0; X!0; S_1] \ \ \| \ \ *[S_1;\ Y!1; S_2] \ \ \| \ \ *[S_2; X!2; S_3] \ \ \| \ \ *[S_0; S_3]$$

When this is implemented as a circuit, there is now a wiring conflict. Both X actions are on the same channel, i.e. there is a unique pair of request and acknowledge wires

between the process and its environment, defining this channel. If two C-elements are instantiated, each driving the request/acknowledge wire, this would result in multiple drivers existing for the same node, which is incorrect. In order to rectify this, the input CHP needs to be rewritten into a form where each channel is accessed at most once in a single iteration. Syntactically, this is equivalent to every channel appearing at most once in the CHP description of the process.

This is achieved with the help of fresh channels that act as aliases for the original channel. For the example above, this can be achieved by introducing aliases X_1 and X_2. In order to correctly manage accesses of these new channels and the single old channel, a corresponding handler process is also required. One might assume that the following is a correct rewrite of the above program since each channel access appears only once:

$$[X_0!0;\ Y!1;\ X_1!2] \quad \|$$
$$s := 0;\ \ *[[s = 0 \longrightarrow X_0?x, s := 1 [] s = 1 \longrightarrow X_1?x, s := 0];\ \ X!x]$$

However, this has the flaw of adding slack on the X channel and is thus semantically different from the original program. In order to see this more clearly, consider the following closed system:

$$*[X!0;\ Y!1;\ X!2] \ \| \ ([\,|\,\overline{X} \longrightarrow good [] \overline{Y} \longrightarrow bad\,|\,]...)$$

Here, the bad program can never get executed as the non-deterministic selection will always pick the only branch whose guard is true, i.e. the first one. However, the rewritten system would be:

$$*[X_0!0;\ Y!1;\ X_1!2] \quad \|$$
$$s := 0;\ \ *[[s = 0 \longrightarrow X_0?z [] s = 1 \longrightarrow X_1?z];\ X!z,\ \ s := 1 - s] \quad \|$$
$$([\,|\,\overline{X} \longrightarrow good [] \overline{Y} \longrightarrow bad\,|\,]...)$$

In this program, bad has a possibility of getting executed, which is incorrect. In order to fix this, we modify the structure of the handler process, by forwarding the value from the

channel to the environment first and delaying the completion of the handshake. This tightly couples the channel actions on the original (X) and alias (X_0, X_1) channels, resulting in zero slack addition. For the example above, the correct handler process would be:

$$
\begin{array}{l}
*[X_0!0;\ Y!1;\ X_1!2] \quad \| \\
s := 0;\ \ *[[s = 0 \longrightarrow [\overline{X}_0 \longrightarrow z := X_0]\ \ [] s = 1 \longrightarrow [\overline{X}_1 \longrightarrow z := X_1]];\\
\qquad\qquad X!z; [s = 0 \longrightarrow X_0?[] s = 1 \longrightarrow X_1?];\ \ s := 1 - s] \quad \| \\
([\,|\,\overline{X} \longrightarrow \mathit{good}\,[]\,\overline{Y} \longrightarrow \mathit{bad}\,|\,]...)
\end{array}
$$

The assignment of a variable to a channel ($z := X_1$) stores the value pending on the channel (and we know there is one pending for sure since the probe must have evaluated to true) into the variable, without performing a handshake on it. Later, after forwarding the value on X, we complete the handshake on the appropriate alias channel. The reader can verify that with this handler process, bad can never execute, thereby preserving the semantics of the original program. This has the added benefit, due to latch elimination, of the handler process never latching the data on the channel and purely forwarding it from the original process.

The same problem arises in the case of a multiple receive as well, and the solution is essentially the same. As a simple example, the correct rewrite for $*[X?x; X?y]$ is:

$$
\begin{array}{l}
*[X_0?x; X_1?y] \quad \| \\
s := 0; *[[\overline{X} \longrightarrow z := X]; [s = 0 \longrightarrow X_0!z [] s = 1 \longrightarrow X_1!z]; X?, s := 1 - s]
\end{array}
$$

The case of linear programs is straightforward, since the ordering of the accesses of the fresh alias channels can be statically determined. However, once control flow is introduced through the use of selections and loops, there is extra information that needs to be communicated to the handler process in order for correct handling. To see how this works, consider the following program fragment:

$$
\begin{array}{l}
*[..;\ \ A?x;\ \ ..; \\
\quad [G_1 \longrightarrow [G_{11} \longrightarrow ..[]G_{12} \longrightarrow A?x];.. \\
\quad []G_2 \longrightarrow ..];\ \ ..; \\
\quad *[\ ..;\ \ A?x;\ \ ..\ \ \longleftarrow\ \ G_3];\ \ ..;\ \ (p) \\
\quad [G_4 \longrightarrow ..[]G_5 \longrightarrow ..];\ \ ..]
\end{array}
$$

where all accesses of A are explicitly stated, and a program fragment (..) denotes arbitrary CHP that does not contain accesses of A. Here, after replacing A with freshly generated aliases, there is no sequencing on accesses of these aliases that can be determined without actually executing the program, since the sequence of channel accesses is data-dependent. Hence, the handler process needs information about the execution state of the original program. To be precise, the handler requires information about which branches were taken in the program.

However, notice that complete state information does not have to be transmitted. Intuitively, it is obvious that if there are selections/loops where the channel is not accessed at all, then the choice of branch in that particular selection/loop is inconsequential to the channel access handler. In the example above, once the program is at point p, the handler does not need to know whether G_4 or G_5 was true, since there are no accesses of A within that selection and regardless of which branch is taken, the next access will occur at the one marked in red.

In order to generate the handler process, we first perform a pre-processing step, which involves building a state table from the CHP program. A CHP program, like any other program, is a directed graph, with vertices representing elementary actions and edges representing control flow; and translating this into a state transition table is straightforward to achieve. The annotated CHP, with channel accesses of A replaced by accesses on freshly generated alias channels $\{A_i\}$ is shown below:

Table 3.1: State Transition Table for CHP with Multiple Channel Accesses for Channel A, before (left) and after (right) optimization .

Current State	Condition	Next State
1	G_1	8
1	G_2	5
8	G_{11}	9
8	G_{12}	2
9	$True$	4
2	$True$	4
5	$True$	4
4	$True$	3
3	G_3	3
3	$\neg G_3$	6
6	$True$	7
7	$True$	1

Current State	Condition	Next State
1	G_1	8
1	G_2	3
8	G_{11}	3
8	G_{12}	2
2	$True$	3
3	G_3	3
3	$\neg G_3$	1

$*[..;\ \textcircled{1} A_1?x;\ ..;$

$[G_1 \longrightarrow \textcircled{8}[G_{11} \longrightarrow \textcircled{9}..[]G_{12} \longrightarrow \textcircled{2} A_2?x];..$

$[]G_2 \longrightarrow \textcircled{5}..];\textcircled{4}\ ..;$

$*[\ \textcircled{3}..;\ A_3?x;\ ..\ \longleftarrow\ G_3];\textcircled{6}\ ..;$

$[G_4 \longrightarrow ..[]G_5 \longrightarrow ..];\textcircled{7}\ ..]$

where the numbers in red encode the state of the program. A state transition table that corresponds to this is shown in Table 3.1 (left). For convenience, the state encoding is chosen such that states $\{1, .., n\}$ are the states where an alias access occurs, but this is not necessary for correctness. The condition $True$ is used to denote an unconditional state transition.

The state transition table can be optimized as follows to remove redundant transitions. Consider state 5, where there is no alias access. The only possible state transition from here is an unconditional one to state 4. Hence, the seventh row in Table 3.1 (left) can be removed and the second row can be modified such that the resulting state for that row is 4 instead of 5. Precisely the same optimization can be applied to the transition from 6 to 7,

and several others. This is in effect, computing the transitive closure of the unconditional state transitions, when the source state is one where an alias access does not occur. The final optimized state table is shown in Table 3.1 (right), and the corresponding annotated CHP is shown below:

$$
\begin{array}{l}
*[..;\ \ \textcircled{1} A_1?x;\ \ ..; \\
\quad [G_1 \longrightarrow \textcircled{8}[G_{11} \longrightarrow ..[]G_{12} \longrightarrow \textcircled{2} A_2?x];.. \\
\quad []G_2 \longrightarrow ..];..; \\
\quad *[\ \textcircled{3}..;\ \ A_3?x;\ \ ..\ \ \longleftarrow\ \ G_3];\ \ ..; \\
\quad [G_4 \longrightarrow ..[]G_5 \longrightarrow ..];\ \ ..]
\end{array}
$$

Once this optimized state encoding and transition table have been generated, the handler process can be generated. We simply introduce communications of branch decisions on fresh channels and construct the handler process as a state machine:

$*[..;\ \ A_1?x;\ \ ..;\ \ C_{g1}!(G_1?\ \ 0:1);$

$\quad [G_1 \longrightarrow C_{g2}!(G_{11}?\ \ 0:1);$

$\qquad\qquad [G_{11} \longrightarrow .. [] G_{12} \longrightarrow A_2?x];..$

$\quad [] G_2 \longrightarrow ..];\ \ ..;$

$\quad *[\ ..;\ \ A_3?x;\ \ ..;\ \ C_{g3}!(G_3?\ \ 0:1) \longleftarrow\ \ G_3];\ \ ..;$

$\quad [G_4 \longrightarrow .. [] G_5 \longrightarrow ..];\ \ ..]$

$\|\ \ s := 1;$

$*[\ [s = 1 \vee 2 \vee 3 \longrightarrow [\overline{A} \longrightarrow x := A] [] else \longrightarrow skip];$

$\quad [s = 1 \longrightarrow A_1!x$

$\quad [] s = 2 \longrightarrow A_2!x$

$\quad [] s = 3 \longrightarrow A_3!x$

$\quad [] s = 8 \longrightarrow skip$

$\quad];$

$\quad [s = 1 \vee 2 \vee 3 \longrightarrow A? [] else \longrightarrow skip];$

$\quad [s = 1 \longrightarrow C_{g1}?c; nxt := (c = 0)?\ \ 8:3$

$\quad [] s = 2 \longrightarrow nxt := 3$

$\quad [] s = 3 \longrightarrow C_{g3}?c; nxt := (c = 0)?\ \ 3:1$

$\quad [] s = 8 \longrightarrow C_{g2}?c; nxt := (c = 0)?\ \ 3:2$

$\quad];$

$\quad s := nxt\ \ ]$

The reader can verify that these two process in parallel do indeed implement the required behavior. Note that in practice, we actually re-encode the states $\{1, 2, 3, 8\}$ as $\{0, 1, 2, 3\}$ since this would enable us to represent the s and nxt variables with only two bits. In case several channels are accessed more than once, there are more handler process that need to be generated, one per channel that is accessed multiple times. The entire procedure described here is performed automatically as a pre-processing step, before circuit

synthesis. Note that this must be done before the extraction of loops described in Section 3.8.

3.10 Latch Minimization

The datapath synthesis strategy described earlier was built upon the simple assumption that every assignment of or receive into a particular variable must result in the generation of a storage element - a pulsed latch or QDI register depending on the chosen circuit family. While evidently correct, this results in far more storage elements than the minimum required to correctly implement a given CHP program. Consider the simple CHP program:

$$*[L_1?x; L_2?y; y := y + x; R_1!y; L_3?z; R_2!(z + y)]$$

Here, according to the STF-style datapath, there are four sets of latches generated as there are four assignments/receives to variables in the program. However, notice that the latch for the statement $y := y + x$ produces only a transient value that is used to compute values later in the program. Hence, the latches for this assignment can be replaced by wires, thereby significantly reducing the circuit complexity. The next natural question that arises is whether this sort of determination, about when an assignment can simply be wires instead of an actual latching, can be performed statically for all CHP programs. It is easy to say for linear programs, but is the same also true in the presence of complex control flow?

Recall that after handling nested loops and multiple channel accesses, as described in Sections 3.8 and 3.9, synthesizable programs always consists of a set of initial conditions and a single loop that wraps an acyclic CHP fragment, i.e. it is of the form $\{x_i := v_i\}; *[true \rightarrow P]$ where the control flow inside P is acyclic. Hence, for every use (read of the value held in it by an expression) of a variable in P, we can iteratively trace its origin backwards until we either arrive at a receive or at an initial condition. Intuitively,

this means that every value that is computed in a CHP program of this form must have come from a receive or from the value of another variable from the end of the previous iteration of the loop.

This means that it is unnecessary to generate latches for every assignment of a variable in the CHP, and instead it is sufficient to generate latches only for all the receives and all the loop-carried variables. All other internal assignments of the form $x := e$ simply consists of the combinational logic for e, the output of which is exposed in the places where originally the output of the latch for x would be exposed. The original latching strategy resulted in a number of latches (N_L):

$$N_L = \sum BW(C?x) + \sum BW(x := e) + \sum BW(x_i := v_i)$$

i.e. the sum of (in order) bitwidths of receives, assignments and initial conditions, where the second term usually dominates for programs of non-trivial sizes. This penalized computing intermediate values, which are necessary for writing readable code. The new latching strategy eliminates the second term entirely, and applying this optimization results in an improvement of the generated circuits across all metrics as several points in the program that were originally data store operations that required waiting for the latch to complete writing are now essentially instantaneous. This change also has the additional benefit of simplifying the control element for an assignment greatly. The original controller consisted of a 2-input C-element and 2 inverters. This can now simply be replaced by wires as well, making the controller zero-cost. As an example, consider the two CHP programs:

$$*[A?x; B?y; (t := x; x := y; y := t); C!(x - y)]$$

$$*[A?x; B?y; C!(y - x)]$$

In the original datapath synthesis strategy, the circuit for the first program would contain 3 additional sets of latches relative to the second, but the new strategy would result in

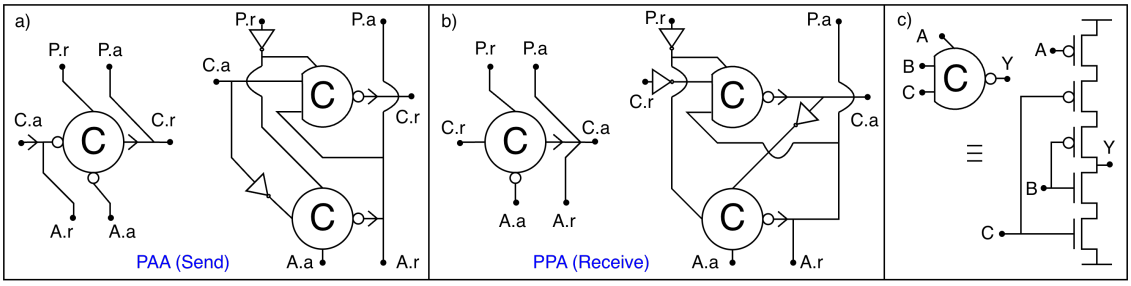


Figure 3.12: Original (left) and improved (right) control elements for sends (a) and receives (b) in the four-phase C-element-based control circuit family. The new elements allow the reset half-phase of the channel handshake to happen in parallel, and consequently improve the cycle time of the circuit. c) Implementation of the asymmetric C-element used in the new controllers.

exactly the same circuit for both programs. The intermediate assignments in the first program would get synthesized purely into crossover of wires, with no additional gates being generated.

3.11 Optimized Control Elements

The original implementations of the two elements (send, receive) were chosen to be the cheapest possible in terms of transistor count. However, this has the drawback of not allowing the channels to reset completely in parallel. In the original circuits, multiple processes that are interconnected end up having their reset phase synchronized with each other as a consequence of the controller structure. This results in added latency that can be avoided by allowing the reset to happen in parallel. The new implementations of the send (PAA) and receive (PPA) elements are shown in Fig. 3.12. The ports labeled P and A connect to the previous and next elements in the ring, while the port labeled C is the channel port.

With these circuits, the channel can complete the reset phase (i.e. resetting the request and acknowledge wires to zero) in parallel with the execution of the remaining elements in the ring. Note that these elements are used only for the 4-phase datapath where the parallel

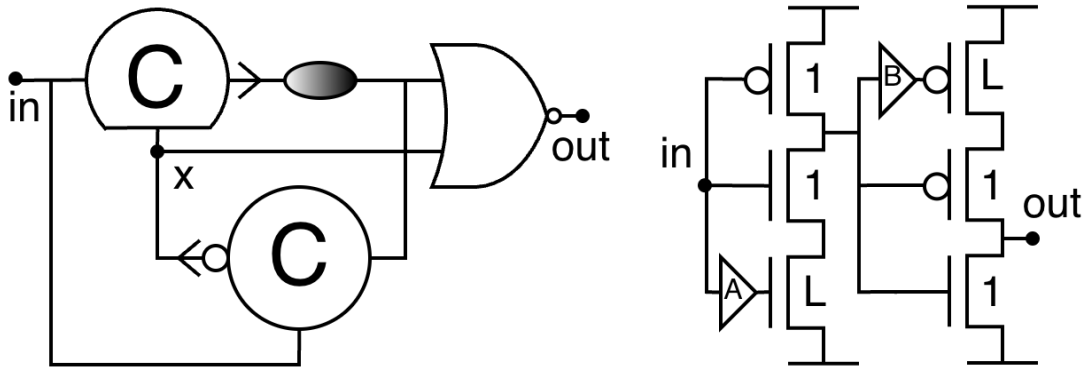


Figure 3.13: Asymmetric delay lines that propagate a rising edge slower than a falling edge. Left: Double reuse of a symmetric delay to emulate an asymmetric delay. Right: Recursively constructed delay line that uses asymmetrically sized inverters.

reset completion is desirable. In the 2-phase datapath, there is no explicit reset phase, and as such, the original elements are efficient. Although the sizes of the controllers are now increased, this implementation actually exhibits lower latency due to distribution of load across the two gates. Further, the overall controller overhead for the entire program is now greatly reduced due to latch minimization. Originally, the number of controller gates was proportional to the number of actions in the program, i.e. the sum of the number of receives, sends and assignments. With the improved method, it is now proportional to the number of channel accesses, which is typically a much smaller number.

3.12 Asymmetric Delay Lines

The delay lines in the synthesized four-phase circuits are required to match the combinational logic delay only in the positive half-phase of their operation. In the negative half-phase, it is not required and ideally we would like for the reset half-phase to complete as quickly as possible. Thus, in order to allow the 4-phase circuits to achieve their maximum possible performance, asymmetric delay lines, i.e. ones which propagate a rising edge (1) and falling edge (0) at different rates, are required. Hence, we are interested in delay lines that propagate a rising edge much slower than a falling edge. There have been

several proposed solutions for this [42, 43]. Fig. 3.13 shows two topologies that allow a large rising edge delay and the shortest possible falling edge delay, that of 1 NMOS and 1 PMOS switching. In the topology on the left, the production rule for the asymmetric C-element is:

$$in \wedge x \rightarrow y\uparrow$$
$$\neg x \rightarrow y\downarrow$$

which can trivially be made CMOS-implementable. Initially, x is high, out is low, and the rising edge on in caues the gate to fire. After going through the delay line once, x is pulled low by the bottom C-element firing. After another trip through the delay line, the output of the NOR-gate goes high, effectively using the delay line twice to generate a large delay. When in goes low, out goes low after merely two gate firings, resulting in an extremely short delay. This strategy is quite area-efficient for large delays as the internal delay line only needs to be half as long as if a simple delay line was used.

The topology on the right is a recursively constructed asymmetric delay line, which internally uses the same topology as a sub-circuit (the elements depicted as buffers, A and B, are themselves the same delay line). In order to understand how this works, consider the base case, where the buffers are just wires, and the delay line is asymmetric due to the sizing. All transistors have the same width, with the relative lengths as marked in the diagram. This propagates a rising edge slower than a falling edge and forms the base case for A in the recursion. There also exists the dual, which propagates a falling edge slower than a rising edge, and forms the base case for B in the recursion. Now, to construct the next recursive one, we simply use these asymmetric delay lines as A and B. The process can then be performed repeatedly to get delay lines of increasing one-sided delay while restricting the delay of the opposing edge to just 2 transitions (1 minimum-size PMOS, and 1 minimum-size NMOS). The intermediate node would also require a staticizer as there is a period of time where it is undriven. The additional minimum-size FETs in series

with the long FETs is to prevent a transient short, during the period where the slow wave is traveling through A or B, that would result in unnecessary power dissipation.

We incorporate these delay lines topologies in the circuit library alongside existing ones and pick between them automatically during circuit construction, as the range of attainable delays is different for each topology.

3.13 Edge-Triggered Datapath

The first implementation of the datapath relied on pulsed latches. While efficient in terms of area, pulse generators are sensitive to loading effects and it is helpful to have edge-triggered elements as an option, as they void the need for a pulse generator. Fig. 3.14 shows an edge-triggered storage element that effectively looks like a D flip-flop, with the added benefit of eliminating the hold-time constraint. In a conventional D flip-flop, the hold-time constraints arises from the fact that there is a short window of time where both latches are transparent. This creates a transient combinational loop, which enforces the requirement that the minimum delay around the loop must be larger than the time window for which these latches are open. The controller eliminates this by changing the sequence of transitions such that the two latches are never open simultaneously. The hold time requirement is replaced by the internal delay that ensures that the two latches are both closed for a small window of time. On assertion of write request, the first latch closes, followed by the second opening, which captures the data on the input. This is identical to an ordinary D flip-flop. However, on de-assertion of the write request, the sequence is flipped, with the second latch closing before the first opens, as can be seen by the sequence of transitions on the OR- and AND-gates. Hence, there is no instant of time with both latches open. Note that the transitions in the controller are strictly sequenced and this scheme is is actually insensitive to the delay of the OR- and AND-gates.

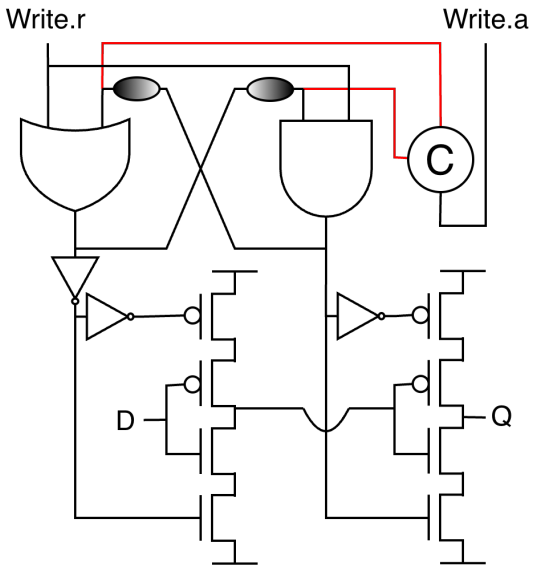


Figure 3.14: Edge-triggered data capture element which consists of two latches connected in series, with a controller to open and close them in the correct sequence.

3.14 Small Program Optimization

The general synthesis technique described so far essentially involves taking a CHP program and instantiating a network of handshake elements where the connectivity closely mirrors the control flow in the program. Recall from Section 3.3 that, for linear programs, we essentially represent a program as multiple programs of the form $*[S_k; C_k; S_{k+1}]$ (where S_k is a passive synchronization and S_{k+1} is an active synchronization) running in parallel, with a special ITB to sequence them correctly. If we apply this to a program of the form $*[L?x; R!f(x)]$, this would result in 3 control elements in total, one for the receive, one for the send and one ITB to close up the ring. However, notice that this program is already of the form that is implemented by a *single* control element, with the middle action i.e. C_k being a skip. Thus, we can accommodate this program on a single control PAA element, greatly reducing the controller complexity.

The same logic can be applied to show that the source and sink controllers can also derived by replacing synchronizations with skips. If only one passive synchronization is

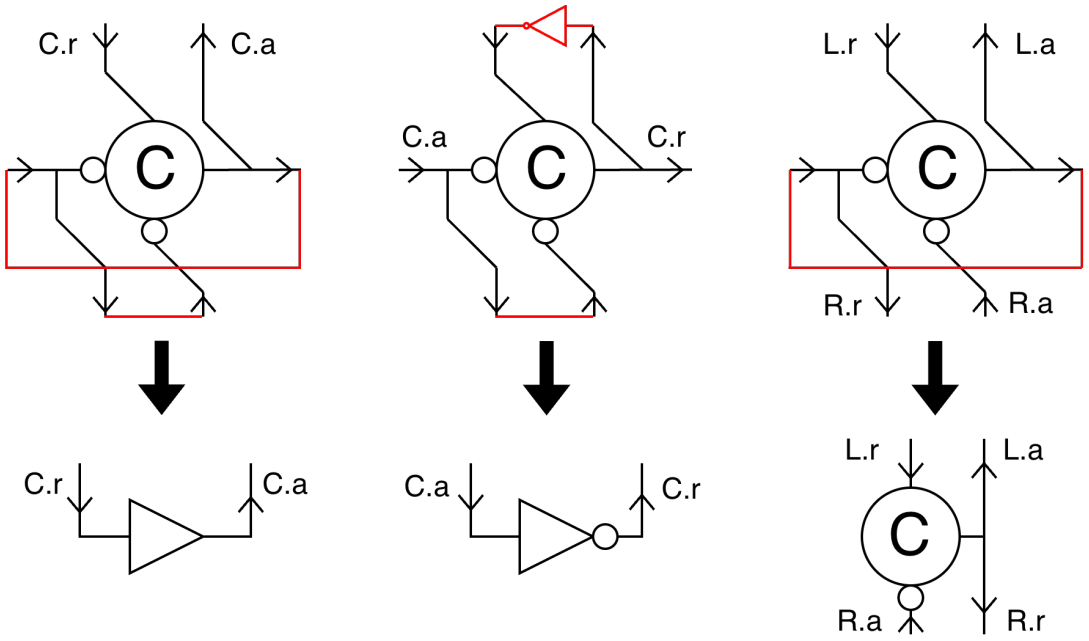


Figure 3.15: The emergence of the sink, source and function-block controllers through termination of some ports in the PAA element.

left, then we end up with a sink. If only one active synchronization is remaining, we end up with a source. Fig. 3.15 shows how the termination of ports, i.e. replacement by a skip, results in the emergence of the simplified controllers.[2] Note that for active synchronization ports, termination involves connecting a passive counterpart, which is represented by just a wire. For passive synchronization ports, termination involves an inverter. It is interesting to see that the function-block controller is identical to the C-element FIFO controller that forms the basis of Sutherland's micropipeline [15].

[2]The transformation involves deletion of a self-invalidation connection from the output of a C-element to its input, which is always valid.

3.15 Experimental Results

3.15.1 First Implementation

The first implementation of Maelstrom did not contain a few key features/optimizations described earlier: latch minimization, optimized controllers and asymmetric delay lines. Nevertheless, we compare this implementation against the Balsa synthesis system. As mentioned before, for combinational logic that implements datapath arithmetic, we use the ABC logic synthesis system for our results [44]. We use the standard combinational synthesis script *resyn2* [45], provided with ABC. We also support the open-source Yosys [46] tool (which uses ABC under the hood) and Cadence's Genus synthesis system as alternatives for combinational logic synthesis. Since we use a single-rail bundled datapath and match the delay in the control path using delay lines, the output circuits from ABC are allowed to glitch, as long as they settle within the time constraint imposed by the delay line. For comparison, we synthesize equivalent Balsa programs to Verilog using the Balsa synthesis system [9], and generate a SPICE netlist from the generated Verilog output. Despite Balsa's age, it is the latest complete general-purpose (i.e. supporting slack elastic and inelastic programs) logic synthesis tool for asynchronous circuits in the literature. We also compare against `chp2prs`, an existing naïve SDT method for translating CHP to circuits.

We report cycle time (a.k.a. cycle period, the inverse of throughput) [47–49] and energy-per-cycle metrics from pre-layout SPICE simulations. We also report layout area from a placed, power-routed and global-routed design of 100 instances of each circuit. The reported number is an average, per-instance area, across the 100 instances. The results of our comparison are summarized in Table 3.2, and we use the geometric mean to compare normalized metrics (where 1.0 corresponds to Maelstrom). Across our test cases, on average, our synthesis method shows an average improvement of 39% in area, 50% in cycle time, and 58% in energy-per-cycle over the Balsa synthesis method. We would

also like to note that the test case complexity is somewhat limited since Balsa-generated Verilog netlists for more complex programs often displayed incorrect functionality, and these issues could not be resolved due to Balsa no longer being an active project.

The results from Maelstrom 2-phase are in line with what would be expected, with approximately half the cycle-time (0.52×) and energy-per-cycle (0.55×) of Maelstrom. The area for Maelstrom 2-phase is slightly higher due to the added circuit complexity of activating the datapath in both phases (1.04×). Balsa's sequencer optimizations preclude this natural extension to 2-phase datapath activation, as opposed to the (unavailable) Haste/TiDE tools, which could be configured to produce 2-phase circuits. The primary reason that the SDT approach is outperformed by Maelstrom is the difference in the core sequential synthesis algorithm (Section 3.2). SDT works by having a statement request the execution of a sub-statement, wait for its completion, and then report that the statement is complete [32]. Applying this repeatedly leads to a "telescoping" effect—the execution of nested statements are "telescoped" by the outer statements waiting for completion. Maelstrom's approach avoids this overhead, leading to better cycle time.

The cycle time (and therefore the throughput) of an asynchronous circuit that is synthesized by a sequential method depends on the exact input CHP specification. For example, consider the two following programs:

$$*[L_1?x_1; L_2?x_2; [x_1 > x_2 \longrightarrow y := (x_1 - x_2); R!y \;[]\; else \longrightarrow y := (x_2 - x_1); R!y]]$$

$$*[L_1?x_1, L_2?x_2; R!((x_1 > x_2)?(x_1 - x_2) : (x_2 - x_1))]$$

It should be clear that from an I/O perspective, both perform the same operations—that of calculating the absolute difference of the two inputs. But the first version evaluates $(x_1 - x_2)$ and $(x_2 - x_1)$ conditionally, unlike the second one that computes both. When synthesized, the circuit that is produced for the first program is much more expensive than

Program	CHP	Method	Area		Cycle Time		Energy	
			μm^2	Ratio	ns	Ratio	pJ	Ratio
Buffer	$*[L?x; R!x]$	Maelstrom (2-phase)	**299** (310)	- (1.04)	0.35 (**0.22**)	- (0.61)	0.10 (**0.06**)	- (0.60)
		chp2prs	1258	4.22	4.25	12.01	0.86	8.48
		Balsa	510	1.71	1.32	3.73	0.36	3.58
Sequence	$*[L\_1?x\_1; R\_1!x\_1; L\_2?x\_2;$ $R\_2!x\_2; L\_3?x\_2;$ $R\_3!x\_3; L\_4?x\_4; R\_4!x\_4]$	Maelstrom (2-phase)	**1498** (1557)	- (1.04)	2.73 (**1.44**)	- (0.53)	0.61 (**0.40**)	-
		chp2prs	3595	2.83	25.66	9.40	3.60	5.90
		Balsa	2084	1.64	6.22	2.28	2.67	4.38
Parallel	$*[L\_1?x\_1, L\_2?x\_2, L\_3?x\_3,$ $L\_4?x\_4; R\_1!x\_1, R\_2!x\_2,$ $R\_3!x\_3, R\_4!x\_4]$	Maelstrom (2-phase)	**1553** (1620)	- (1.04)	1.06 (**0.56**)	- (0.53)	0.61 (**0.37**)	- (0.61)
		chp2prs	5178	3.32	6.24	5.89	6.97	11.37
		Balsa	2527	1.62	1.94	1.83	1.19	1.94
Adder	$*[L\_1?x\_1,$ $L\_2?x\_2;$ $R!(x\_2 + x\_1)]$	Maelstrom (2-phase)	**1215** (1236)	- (1.02)	2.42 (**1.26**)	- (0.51)	0.35 (**0.23**)	- (0.67)
		chp2prs	2022	1.66	6.38	2.64	2.52	7.28
		Balsa	2760	2.27	3.31	1.37	0.94	2.71
Multiplier	$*[L\_1?x\_1,$ $L\_2?x\_2;$ $R!(x\_2 * x\_1)]$	Maelstrom (2-phase)	**2420** (2470)	- (1.02)	3.66 (**1.91**)	- (0.52)	0.51 (**0.32**)	- (0.63)
		chp2prs	3114	1.29	5.71	1.56	2.68	5.27
		Balsa	-	-	-	-	-	-
Split	$*[C?c; L?x;$ $[c = 0 \rightarrow R\_1!x$ $[]c = 1 \rightarrow R\_2!x]]$	Maelstrom (2-phase)	**684** (711)	- (1.04)	1.45 (**0.73**)	- (0.51)	0.28 (**0.18**)	- (0.66)
		chp2prs	2879	4.21	10.05	6.93	2.85	10.33
		Balsa	1316	1.93	2.33	1.61	0.39	1.42
Merge	$*[C?c;$ $[c = 0 \rightarrow L\_1?x$ $[]c = 1 \rightarrow L\_2?x]; R!x]$	Maelstrom (2-phase)	**1124** (1193)	- (1.06)	1.63 (**1.13**)	- (0.69)	0.30 (**0.15**)	- (0.50)
		chp2prs	2981	2.56	15.14	9.29	2.36	7.87
		Balsa	2051	1.82	2.59	1.59	0.44	1.45
GCD		Maelstrom (2-phase)	**4930** (5072)	- (1.03)	17.22 (**8.93**)	- (0.52)	9.83 (**5.03**)	- (0.51)
		chp2prs	8881	1.81	374.52	21.75	85.60	8.71
		Balsa	6488	1.31	37.64	2.19	28.35	2.89
Fibonacci		Maelstrom (2-phase)	**3921** (4043)	- (1.03)	20.65 (**11.87**)	- (0.57)	8.43 (**4.53**)	- (0.54)
		chp2prs	8150	2.08	212.65	10.30	104.80	12.44
		Balsa	5983	1.53	57.12	2.77	17.33	2.06
Bresenham		Maelstrom (2-phase)	**14275** (14543)	- (1.02)	27.23 (**15.65**)	- (0.57)	18.50 (**9.60**)	- (0.52)
		chp2prs	19276	1.44	312.39	11.47	163.20	8.82
		Balsa	20107	1.51	64.12	2.35	44.80	2.42
Average excluding multiplier	Balsa vs. Maelstrom (1.0)			**1.64**		**1.99**		**2.38**
	Balsa vs. Maelstrom 2-phase (1.0)			**1.58**		**3.84**		**4.30**
	Maelstrom 2-phase vs. Maelstrom (1.0)			**1.04**		**0.52**		**0.55**

Table 3.2: Results From SPICE Simulations of synthesized CHP and equivalent Balsa programs in a 65nm process. Lower values are better across all metrics. Averages are geometric means, with larger numbers corresponding to greater factors of improvement for Maelstrom. Maelstrom 2-phase produces larger circuits but performs better in terms of delay and energy.

that of the second due to the presence of selections, extra assignments, and multiple channel accesses. The circuit for the first (second) program has an area of 7453 (2899) μm^2, cycle time of 6.80 (3.02) ns, and energy per cycle of 2.96 (1.05) pJ. Further, the throughput of an asynchronous circuit is also input-dependent, and varies from cycle to cycle. Hence, the average throughput is the pertinent measure. Note that the delay introduced by the controller is small since it is just a single gate per program statement, and the overwhelming majority of the delay arises from the delay of the combinational logic that implements the computation, except for circuits like FIFOs that have no logic.

Finally, it is important to note that the circuits using Maelstrom are practically on-par with hand-designed dataflow circuits when we examine small designs where it is practical to make the comparison. In particular, the building blocks for dataflow circuits (function, split, merge, copy, initial token, buffer, source, and sink [35, 50]) have the same area/delay/energy compared to those automatically generated by Maelstrom. This means that existing dataflow synthesis tools like Fluid can be implemented using CHP-to-CHP mapping followed by Maelstrom, rather than requiring their own custom circuit library.

3.15.2 Tool Runtime

Table 3.3 summarizes results from running Maelstrom on larger programs. The runtime is dominated by ABC and the interface (I/O) between ABC and Maelstrom, which requires printing expressions for ABC to synthesize. Large expression trees in the fourth test case causes Maelstrom's data-dependency analysis, which is required to instantiate registers correctly, to take up a larger fraction of the runtime.

3.15.3 Current Implementation

The current implementation of Maelstrom incorporates all the optimization mentioned earlier. Table 3.4 contains the key metrics for the synthesized circuits. As before, we report

Design	L-1	L-2	L-3	L-4
CHP Statements	20	31	64	28
Total Runtime (s)	1.59	7.98	6.53	7.83
Maelstrom Runtime (%)	10.17	10.10	11.39	22.99
ABC Runtime (%)	76.72	76.70	75.66	56.01
I/O Runtime (%)	13.11	13.20	12.95	21.00
Area (μm^2)	7081	44731	42238	72063
Cycle Time (ns)	1.12	7.10	34.9	47.2
Energy (pJ)	0.99	6.22	32.37	38.37

Table 3.3: Results From SPICE Simulations of Maelstrom-synthesized circuits for large CHP programs. The runtime of Maelstrom is proportional to the size of the CHP program, while the combinational logic synthesis is typically of a much higher complexity. Hence, it is expecetd that this step dominates the runtime.

layout area, throughput and energy-per-token metrics from pre-layout SPICE simulations. On average, the circuits produced by the optimized implementation have an improvement of 9% in area, 16% in token time and 23% in energy consumption, when compared to the original.

The first number in each cell corresponds to the 4-phase circuits, where the datapath is only activated on one phase of the handshake, with the other used exclusively for reset. These circuits can exploit asymmetric delay lines. The numbers in parentheses are for the 2-phase circuits, where the datapath is activated on both phases of the handshakes that propagate through the ring, instead of just on the positive half-phase. These circuits must use symmetric delay lines as they need to match the datapath delay on both phases. All ratios are relative to the 2-phase value of the new implementation for a given test case, which is taken to be 1 and thus omitted in the column.

The circuits for extremely simple CHP programs, such as those that correspond to dataflow elements are essentially unchanged, as these were already on par with hand-optimized circuits. For larger programs, the effect of the improved control elements and reduction of latches is apparent as the synthesized circuit is smaller, faster and uses less energy per computation.

Program	Method	Area		Cycle Time		Energy per Token	
		μm^2	Ratio	ns	Ratio	pJ	Ratio
Buffer	Original (2-phase)	685 (648)	1.05 (1.00)	0.581 (0.149)	3.89 (1.00)	0.15 (0.07)	2.14 (1.00)
	Optimized (2-phase)	685 (648)	1.05 (-)	0.581 (0.149)	3.89 (-)	0.15 (0.07)	2.14 (-)
	Synchronous	716	1.10	0.540	3.62	0.10	1.42
GCD	Original (2-phase)	10130 (13686)	0.84 (1.14)	43.97 (35.95)	1.46 (1.19)	21.00 (26.40)	1.23 (1.55)
	Optimized (2-phase)	8598 (11970)	0.718 (-)	42.34 (30.02)	1.14 (-)	18.95 (16.95)	1.11 (-)
	Synchronous	10721	0.89	25.10	0.83	18.11	1.06
Fibonacci	Original (2-phase)	5011 (6225)	0.98 (1.23)	54.74 (42.05)	1.98 (1.52)	31.28 (22.61)	2.78 (2.01)
	Optimized (2-phase)	4004 (5068)	0.79 (-)	51.86 (27.51)	1.88 (-)	12.53 (11.25)	1.11 (-)
	Synchronous	4555	0.89	21.23	0.77	12.45	1.10
Matrix Multiply	Original (2-phase)	59731 (65126)	1.01 (1.10)	3.18 (1.97)	1.65 (1.03)	6.80 (4.92)	1.42 (1.03)
	Optimized (2-phase)	53038 (59143)	0.89 (-)	3.04 (1.92)	1.58 (-)	5.92 (4.76)	1.24 (-)
	Synchronous	52531	0.88	1.90	0.99	5.43	1.14
Cross-Correlation	Original (2-phase)	19571 (19487)	1.12 (1.12)	4.62 (2.42)	2.13 (1.12)	3.78 (1.71)	2.86 (1.29)
	Optimized (2-phase)	17475 (17395)	1.00 (-)	4.14 (2.16)	1.91 (-)	2.62 (1.32)	1.98 (-)
	Optimized (2-phase) (sized)	17832 (17523)	1.02 (1.01)	3.72 (1.79)	1.72 (0.82)	2.68 (1.39)	2.03 (1.05)
	Synchronous	22740	1.30	1.75	0.81	1.90	1.43
SerDes	Original (2-phase)	4555 (7205)	0.75 (1.19)	57.30 (29.70)	1.99 (1.03)	40.00 (37.67)	1.14 (1.07)
	Optimized (2-phase)	3733 (6050)	0.62 (-)	53.61 (28.67)	1.86 (-)	34.20 (35.00)	0.97 (-)
	Synchronous	3260	0.53	28.32	0.98	30.10	0.86
Shared Function Block	Original (2-phase)	7794 (10586)	0.77 (1.05)	4.61 (3.04)	1.54 (1.02)	1.01 (1.71)	0.65 (1.10)
	Optimized (2-phase)	7257 (9999)	0.72 (-)	4.46 (2.98)	1.49 (-)	0.84 (1.55)	0.54 (-)
	Synchronous	9898	0.99	2.65	0.89	5.12	3.30
Skipping Multiplier	Original (2-phase)	6773 (7378)	0.98 (1.07)	1.37 (0.95)	1.95 (1.35)	0.99 (0.83)	1.80 (1.51)
	Optimized (2-phase)	6256 (6856)	0.91 (-)	1.23 (0.70)	1.75 (-)	0.73 (0.55)	1.32 (-)
	Synchronous	6789	0.99	0.85	1.21	1.01	1.83
L2 Norm	Original (2-phase)	7257 (7938)	0.93 (1.01)	2.18 (1.20)	2.50 (1.37)	0.36 (0.28)	1.56 (1.21)
	Optimized (2-phase)	6845 (7812)	0.87 (-)	1.63 (0.87)	1.87 (-)	0.27 (0.23)	1.17 (-)
	Synchronous	5958	0.76	1.16	1.33	0.18	0.78
Geometric Mean	Maelstrom 2-phase vs. Optimized 2-phase		1.09		1.16		1.23
	Synchronous vs. Optimized 2-phase		0.90		1.11		1.30

Table 3.4: Results of synthesis using the original and optimized implementations of Maelstrom for CHP, and Yosys for Verilog. Area is from a placed, power- and global-routed design in a commercial 65nm technology. The cycle time and energy numbers are extracted from SPICE simulations. All ratios are relative to the Optimized 2-phase value for a given test case.

3.15.4 Comparing with Synchronous State Machines

In order to compare our synthesis with synchronous synthesis, we write Verilog programs that are behaviorally equivalent to the CHP programs, with channels replaced with standard ready-valid interfaces to implement flow control. In order to synthesize the Verilog, we once again use Yosys to maintain fairness in terms of combinational logic optimization. The synthesized circuits are timed using Cadence Tempus to determine the critical path and thus the maximum clock frequency for the circuit. We use this clock frequency in SPICE simulations to extract energy and throughput metrics.

We see that on most programs, the results from the optimized implementation are comparable with equivalent synchronous state machines. The key to improving the performance of the asynchronous circuits is latch elimination, which obviates several unnecessary delays and latchings. Further, for some programs, where there are fast and slow paths, it is possible for the asynchronous circuit to have a smaller average cycle time than the critical period. As an example, consider the skipping multiplier (which can skip the multiplication if either of the inputs is zero). This corresponds to the following CHP:

$$*[I_1?x, I_2?y; [x = 0 \vee y = 0 \longrightarrow z := 0 \ [] else \longrightarrow z := x*y]; \ \ O!z]$$

If the input pairs contain a high density of zeroes, then the average cycle time of the synthesized circuit for this CHP can be smaller than the delay of the multiplier. However, the equivalent synchronous state machine cannot dynamically adjust its speed based on the data and is thus constrained by the worst-case (critical) delay, which is the multiplier delay. For cases such as this, the asynchronous circuit outperforms the synchronous one. The same effect is also observed in the L2-Norm program, which is just a sequence of square-and-accumulate operations. Here too, the asynchronous circuit outperforms in the presence of sparsity.

The final missing pieces that are required for the asynchronous circuits to close the

remaining gap in performance are physical optimizations, such as gate-sizing and buffer insertion. In order to substantiate this claim, we perform gate-sizing manually for one of the test programs (cross-correlation) and report metrics for the sized circuit as well. As seen in Table 3.4, this results in the cycle time of the asynchronous circuit being virtually the same as the synchronous one, while still expending lesser energy per token.

Finally, notice that the difference in token time between the synchronous and asynchronous circuits for the Fibonacci and GCD test cases is significantly larger than the rest. This is due to the fact that these are iterative (i.e. multi-cycle) tests. Hence, even though the asynchronous circuit is only slower than the synchronous one by 200-400ps (without gate sizing), this lag adds up over the multiple cycles that it takes for the circuit to produce a single output data token. Thus, the apparent lag gets multiplied by the number of CHP loop iterations taken to compute each token. Hence, in this case, the benefit of gate-sizing would also be proportional to the number of cycles, as this would improve the circuit cycle time by a constant factor, not the time between output tokens.

3.15.5 Choice of Controller Style

Maelstrom supports 2-phase as well as 4-phase style circuits. However, it is not the case that one uniformly dominates the other in terms of performance. This is dependent on the exact nature of the CHP program that is being synthesized. The 4-phase circuit has more complex handshake (controller) elements that allow for parallel reset, while the 2-phase can accommodate a more lightweight controller as there is no explicit reset phase at all. On the other hand, the circuits that implement selections in 2-phase are more complex than in the 4-phase case as they need to, in some sense, 'remember' each branch that was taken and route the control pulse propagating through the selection on a particular iteration. The 4-phase circuit has the luxury of an entire dedicated phase just for reset and thus requires a simpler circuit for implementing selections. In order to see this more clearly, consider

this simple CHP for a merge:

$$*[C?c; [c = 0 \longrightarrow X?x [] c = 1 \longrightarrow Y?x]; Z!x]$$

Suppose that the sequence of values on the channel C is $0, 1, 0$. Now, the 4-phase circuit simply activates the $c = 0$ branch, resets, activates the $c = 1$ branch, resets, activates the $c = 0$ branch, and resets. Hence, we know that the positive-, or data- half-phase is always a rising edge that propagates through the circuit. In the 2-phase circuit for the same program, the circuit must send an active-high control wave through the $c = 0$ branch, do the same to the $c = 1$ branch without affecting the $c = 0$ branch, then send an active-low control wave through the $c = 0$ branch without affecting the $c = 1$ branch. At the end, the circuit is in a different state than when we started, with the control elements in the $c = 1$ branch still holding a high. This orchestration is implemented with XOR-gates and one-bit latches and adds overhead.

In general, the cost of the 4-phase vs. 2-phase circuit depends on the balance between the number of handshake elements (which is proportional to the number of channel actions) and the number (and size) of selections, as one is cheaper in one family and the other is cheaper in the other.

3.16 Other Circuit Families

So far, we have built our synthesis strategy upon the fundamental C-element micropipeline. But, as mentioned earlier, the methodology is agnostic to the underlying circuits that are actually used to build the control-flow ring elements. As long as sequencers, parallel and selection blocks can be built, then the synthesis strategy can be used. In Figs. 3.16 and 3.17, we show the 3-action sequencers for the MOUSETRAP [51] and GasP [52] circuit families. The parallelizers and selection splits and merges can be constructed by following the same reasoning as for the QDI family. These sequencers have the same

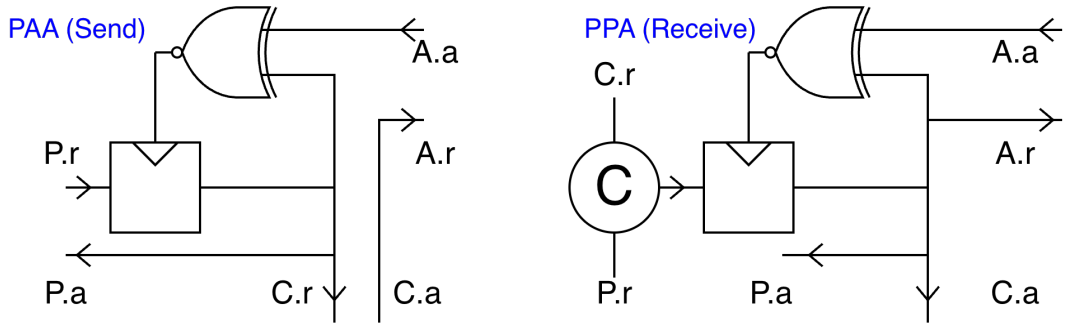


Figure 3.16: The PAA and PPA sequencer elements derived from the MOUSETRAP control family. As before, the three actions that an element tries to sequence are P; C; A.

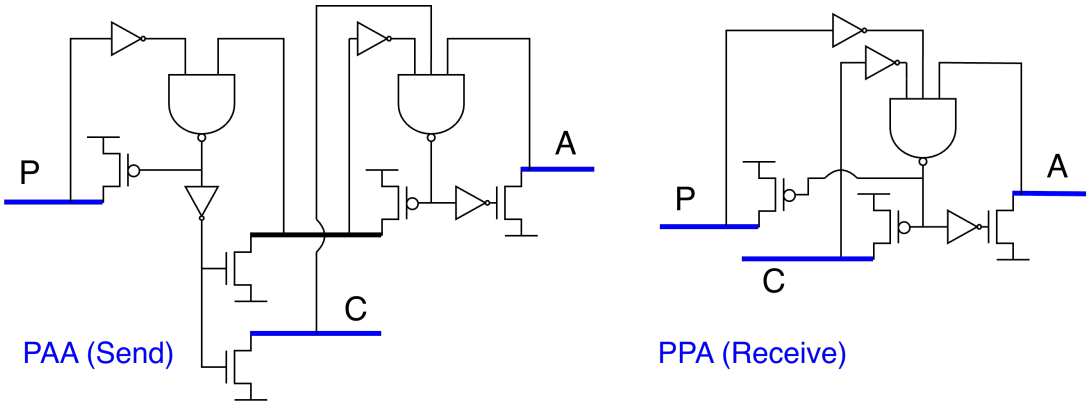


Figure 3.17: The PAA and PPA sequencer elements derived from the GasP control family. As before, the three actions that an element tries to sequence are P; C; A.

timing constraints as the FIFO control circuits that they are derived from. These constraints need to be obeyed in order for correct operation, unlike the QDI control circuits.

For the MOUSETRAP family, the ITB that initializes the ring, and the parallelelizer elements are exactly the same as the ones that are used for the C-element family. The selection split and merge elements are slightly different due to the nature of the controller. Reset is incorporated into the ITB and the latches, which reset with their output low. In a similar fashion, the GasP family sequencers can also be derived from the GasP FIFO control circuit. The sequencer elements and ITB for GasP are significantly different, since this family uses a single wire (referred to as the "state conductor") for sequencing, instead of two (request and acknowledge) used by most control families. Here, ring elements

can be connected in a loop, with all state conductors initially high. The ITB is simply a pulse generator that pulls the first state conductor low when Reset is deasserted. As time progresses, the ring will oscillate without any explicit resetting by an ITB. Again, the selection and parallel blocks can be derived following similar reasoning as before.

3.17 Summary

In this work, we presented a new technique to synthesize behavioral descriptions of asynchronous circuits into gates, and a tool that automatically compiles CHP into these circuits. The approach is agnostic to the underlying control circuit family. The proposed method starts from known high-quality pipeline circuits and uses a simple way to construct sequencer and control-flow elements from them that form a basis set for the synthesis. The method also presents a clear separation between control and datapath circuits, which enables the use of different datapath families. We also compare with synchronous state machines and show that the performance of the asynchronous circuits matches that of the synthesized synchronous logic. The open-source (available via [31]) synthesis tool improves on the existing state-of-the-art asynchronous synthesis techniques by a significant factor, in terms of performance metrics such as energy, delay and area.

Chapter 4

Decomposition

Decomposition is the process of modifying a high-level (CHP) 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 CHP 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 hypergraph. The process of finding these cuts is driven by a novel timing model of CHP, described in terms of RER systems, which were originally introduced to model asynchronous circuits. We derive an RER system that is a simplified, conservative approximation of the circuit that would be produced from the CHP and show how to construct this RER system directly from the CHP instead of synthesizing, constructing the circuit RER system and then simplifying it. 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.

4.1 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 [53, 54], 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 [55]. 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 [18]. Slack elasticity provides theoretical guarantees about the correctness of decomposition techniques that introduce concurrency, including pipelining and projection [56]. Prior work in this area such as the static tokens method [19] 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 [57] 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.

We introduce a hypergraph-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 hypergraph. The framework allows us to formulate the problem of decomposition in terms of finding cuts in a hypergraph that satisfy certain constraints. 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.

4.2 Pre-processing

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 AST, such as dead-code elimination and constant propagation, which have been extensively studied in the software compiler literature [58]. 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:

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

would get rewritten into:

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

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 inelastic CHP programs [18]. 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 (see Section 2.6), which forms the foundation for the data-dependency analysis.

4.3 Dependence-based Decomposition Hypergraphs

Once the AST has been placed into STF, we construct an auxiliary data structure, the dependence-based decomposition hypergraph (DDG), which is the entity that actually forms the basis of the decomposition. The purpose of the DDG is to capture and expose the data-dependencies in the given CHP program independent of the exact control structure in the program.

The vertices/nodes in the DDG correspond to atomic program constructs in the CHP. All nodes in the DDG fall into one of these types:

- 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

Table 4.1: Variables that are used and defined by the different types of nodes in the DDG

Node Type	CHP	Defs	Uses
Receive	$C?x$	x	-
Assign	$x := e$	x	variables in e
Send	$C!e$	-	variables in e
Sel. ϕ	$x := \phi(x_1, .., x_n, x_{g1}, .., x_{gn})$	x	$x_1, ..x_n, x_{g1}, .., x_{gn}$
Sel. ϕ^{-1}	$\{x_1, .., x_n\} := \phi^{-1}(x, [x_{g1}, .., x_{gn}])$	$x_1, ..x_n$	x, $[x_{g1}, .., x_{gn}]$
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	$x_{g_i} := e_{g_i}$ (guard expr.)	x_{g_i}	variables in e_{g_i}

will show later, and is a new feature of our analysis compared to more traditional control- and data-dependency graphs.

- Selection ϕ-node (ϕ), corresponding to a ϕ-function in the STF.
- Selection ϕ^{-1}-node (ϕ^{-1}), corresponding to a ϕ^{-1}-function in STF.
- Loop ϕ-node (ϕ_L), corresponding to a ϕ-function for loops.
- Loop ϕ^{-1}-node (ϕ_L^{-1}), corresponding to a ϕ^{-1}-function for loops.

Each node in the DDG 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 the CHP fragment corresponding to that node. The `uses` is the set of variables whose values are read/used by that node. For example, an assignment node corresponding to $z := f(x, y)$ has the `defs` set $\{z\}$, as z is written to in that statement. The `uses` set for this node is $\{x, y\}$ as values of both of these variables are read/used to compute the expression on the RHS of the assignment. Note that one or both of these sets might be empty. For example, a send action over a channel $C!f(x, y)$ uses $\{x, y\}$ but defines no variables. Similarly, a receive $C?x$ defines x but uses no variables. Table 4.1 specifies exactly the `defs` and `uses` set for every type of node in the graph.

The next step is to define the hyperedges in this hypergraph. The purpose of the hyperedges is to capture the data-dependence between multiple nodes in the DDG. In order

to capture this flow, the natural procedure is to define a hyperedge that starts (has a tail) at a node that defines a variable and ends (has heads) at all the nodes that use that particular variable. Note that the choice of tail node is unique as we are guaranteed that each variable is assigned to precisely once in the program (by virtue of STF).

Thus, we define our DDG as the directed hyper-graph $D : \{V, E\}$, where V is the set of vertices, where each vertex is one of the types specified in Table 4.1. E is the set of directed hyper-edges. Each hyper-edge $e = \{T(e), H(e)\}$ has a singleton set as its tail set $T(e)$ and some subset of V as its head set $H(e)$. The construction of V from the CHP is straightforward. The construction of E is as follows:

- For each variable in the CHP in static-token form, create a hyper-edge e with $T(e)$ being the node that defines that variable, and $H(e)$ being the set of nodes that use that variable.

Most types of nodes in the DDG have at most one hyperedge with that node as the tail, as they define at most one variable. However, the two kinds of ϕ^{-1} nodes are exceptions. The selection ϕ^{-1} defines at most n variables, where n is the number of branches in the selection. The loop ϕ_L^{-1} defines either 1 or 2 variables, based on whether the final value held in that variable at the end of execution of the loop is used or unused.

4.4 Free Decompositions

Once the DDG has been constructed, it is possible that the DDG is comprised of more than one weakly-connected component (WCC). A weakly-connected component of a directed hypergraph is a subset of vertices, such that when considering the directed hypergraph as an undirected hypergraph, every vertex within the subset can be reached from every other within the subset, but cannot be reached from any vertex outside the subset.

Consider the following example:

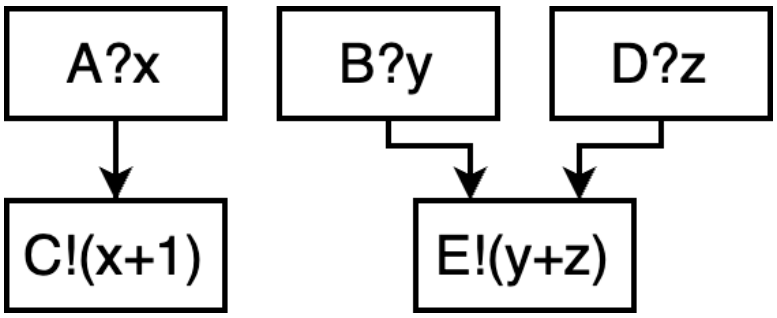


Figure 4.1: Dependence-based Decomposition Hypergraph for the example CHP program. All hyperedges in this DDG happen to be 1-to-1 and thus appear as edges.

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

The DDG for this program is shown in Fig. 4.1. All nodes are basic nodes and the hyperedges are added according to the aforementioned rules. For example, a hyperedge from $B?y$ to $E!(y+z)$ exists since the latter uses a variable that the former defines. Other hyperedges are added similarly. Notice that the DDG for this program consists of two weakly-connected components. The first one contains the nodes for A, C and the second one contains the nodes for B, D, E.

The key observation that allows decomposition is that when this kind of disconnection exists in the hypergraph, each WCC can be written out as a separate process. Intuitively, this happens due to the fact that all the data-dependencies in the program are encoded in the hyperedges and the absence of a path from one WCC to another implies that there is no data-dependence between them and thus can operate fully in parallel.

Here, the original program can be rewritten into two concurrent programs, as follows:

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

Here, the assumption of slack elasticity is crucial. In the original program, output tokens on C are guaranteed to arrive before those on E. This is not true in the decomposed system as those channels now exist in concurrent portions. However, under a slack elastic environment, the relative sequencing of actions on different channels does not need to

be preserved, only the order among tokens on the same channel. Since the decomposed system is uniformly more concurrent than the original, it is guaranteed to not deadlock with any slack elastic environment.

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

4.5 DDG Cuts

A cut is a transform applied to the DDG D, which is defined as follows. A cut is composed of two steps:

- Node additions to V
- Hyper-edge transformations and additions to E

Consider a variable x, with corresponding hyper-edge e. In the first step, the cut transforms the vertex set as: $V \to V \cup \{s_x^C, r_{x'}^C\}$, where s_x^C is a fresh vertex corresponding to a send ($C!x$) of the variable x over a freshly introduced channel C. $r_{x'}^C$ is the corresponding fresh receive ($C?x'$) vertex over the same channel into a fresh variable x'.

In the second step, e is transformed as $\{T(e), H(e)\} \to \{T(e), H_x(e) \cup \{s_x^C\}\}$, where $H_x(e)$ is some subset of $H(e)$. Finally, a new hyper-edge e' (corresponding to the fresh variable x') is added to E, which is defined as $\{T(e'), H(e')\} = \{r_{x'}^C, H(e) \backslash H_x(e)\}$. $H_x(e)$ and $H(e) \backslash H_x(e)$ essentially form a partition of $H(e)$.

This process is also referred to as copy-insertion, as we have essentially created a copy of x, i.e. x', inserted a send-receive pair corresponding to this copy: $C!x, C?x'$, and updated the data dependencies such that a part of the program (the complement of the

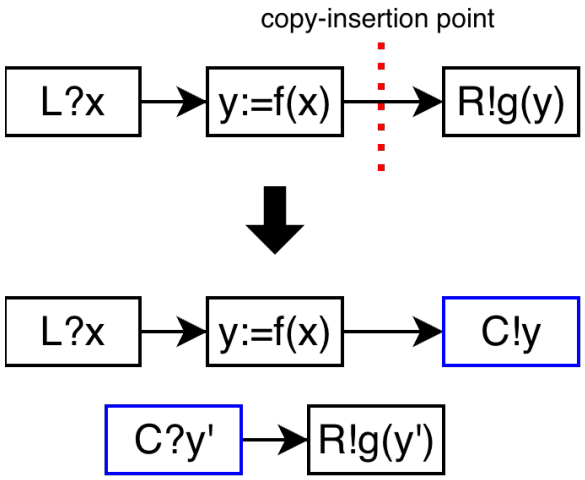


Figure 4.2: DDG of the example CHP, demonstrating copy-insertion.

chosen subset $H_x(e)$ of $H(e)$) that depended on x now depends on its copy x'. Performing copy-insertion allows us to splinter a CHP program into multiple fragments which can then execute in parallel, with necessary data dependencies being transmitted over fresh channels between the fragments.

As a trivial example, consider: $*[L?x; y := f(x); R!g(y)]$. Constructing the DDG (Fig. 4.2) and performing copy-insertion on the hyper-edge corresponding to y (the subset choice is trivial since there is only a single node in the head set), results in the DDG decomposing into two disconnected components. Due to the lack of data dependencies between the two pieces, they can now be executed entirely in parallel, resulting in: $*[L?x; y := f(x); C!y] \parallel *[C?y'; R!g(y')]$. The hyperedges in the DDG can still be cut further. However, this is not useful in this case. The only result of performing this would be to add buffering on the channels L, C, R. The reader can verify this quite easily.

Apart from this kind of straightforward data-dependence based decomposition, the DDG also allows for selections to be fractured. This is enabled by the use of the guard nodes. In order to illustrate this, consider the following example:

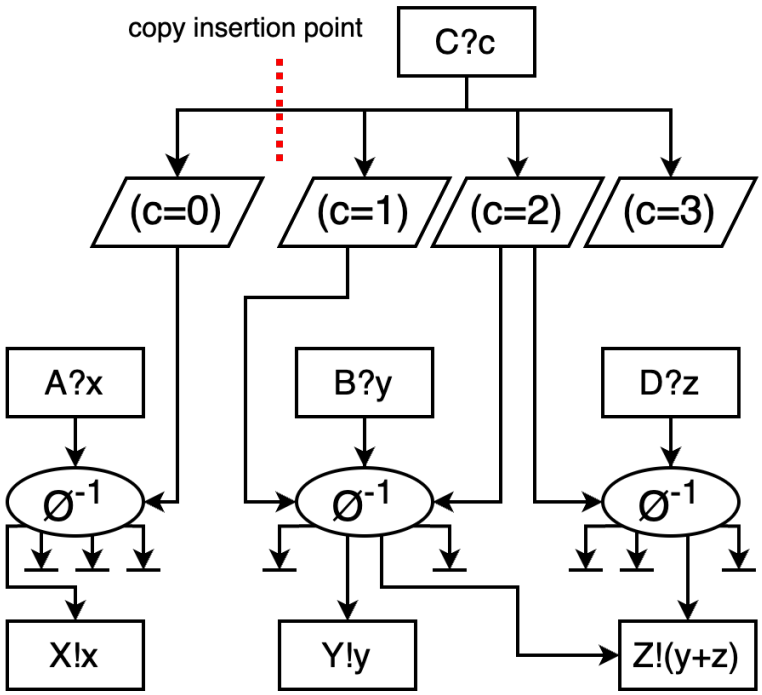


Figure 4.3: Dependence-based Decomposition Hypergraph for the example CHP program with disjoint sets of variables in selection branches. Null-terminated hyperedges do not actually exist in the graph as they do not define a variable, and are shown only for clarity.

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

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. 4.3, is c, which is used to compute the guards. If copy insertion is performed on the hyperedge leading into the guard nodes such that $c = 0$ is in one paritition and the rest are in another, then the DDG can be split into two components. When the processes are reconstructed, the resulting CHP is:

$*[A?x, C_1?c_1; [c_1 = 0 \longrightarrow X!x \ [] else \longrightarrow skip] \ ] \ \|$

$*[B?y, D?z, C?c; C_1!c;$

$\quad [c = 1 \longrightarrow Y!y [] c = 2 \longrightarrow Z!(y + z)$

$\quad [] c = 3 \longrightarrow skip [] else \longrightarrow skip] \ ]$

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. Note that it is the presence of guard nodes here that exposes the fact that this selection can be splintered into two pieces. The decomposed system is, in effect, the result of projecting [16] the program appropriately.

The cut operation described here forms the basis of our optimization problem. All possible decompositions of the process can be expressed as different cuts on this hypergraph. One particular cut choice would result in something familiar. Consider the case where every single hyperedge in Fig. 4.3 is cut until there is no use in cutting it further (i.e. its head set is a singleton and thus there exists only one possible partition, resulting in simple FIFO insertion). This would result in the following CHP:

$*[C?c; C_0!c, C_1!c, C_2!c] \ \|$

$*[A?x; A_1!x] \ \| \ *[B?y; B_1!y] \ \| \ *[D?z; D_1!z] \ \|$

$*[A_1?x_1; C_0?c_0; [c_0 = 0 \longrightarrow X!x_1 [] else \longrightarrow skip]] \ \|$

$*[B_1?y_1; C_1?c_1; [c_1 = 1 \longrightarrow Y!y_1 [] c_1 = 2 \longrightarrow T!y_1 [] else \longrightarrow skip]] \ \|$

$*[D_1?z_1; C_2?c_2; [c_2 = 2 \longrightarrow U!z_1 [] else \longrightarrow skip]] \ \|$

$*[T?y_2, U?z_2; D_1!(z_2 + y_2)]$

Upon observation, every process is essentially a simple dataflow element. Thus, the most fine-grained decomposition of the DDG actually results in the dataflow decomposition of a CHP program. Strictly, the $(else \rightarrow skip)$ branch in the selections in the processes above would actually be sends on channels that simply connect to sinks and throw away

the token on that channel. We omit that for brevity. In the event the original DDG has cycles, we can break all loops with the introduction of initial token buffers (see [19, 55]); 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.

The techniques described above show that the DDG-based decomposition is a generalization of other decomposition techniques such as projection and dataflow. By controlling how many hyperedges are cut and to what degree, we can control the amount of pipelining that is achieved. Each cut adds some hardware cost — that of the circuits needed to implement the send-receive pair ($C!x \parallel C?x_1$) for that particular cut. In return, it perhaps provides some degree of pipelining by breaking up the DDG into more than one weakly-connected component. Now that we have formalized the problem of finding the best decomposition as choosing hyperedges to cut, we can attempt to derive algorithms for making this choice.

4.6 SCC Collapse

The programs we have considered so far do not have complicated loops and cyclic data-dependencies. However, this tends to be common in most useful CHP programs. As an example, consider:

$v := 0;$

$*[[v = 0 \longrightarrow A?v$

$\quad []\, else \longrightarrow v := v - 1];$

$\quad [v > 2 \longrightarrow X?x;\ Y!(x/v)$

$\quad []\, else \longrightarrow P?p;\ Q!(p*v)]]$

In this program, the variable v is a loop-carried dependency for the entire process.

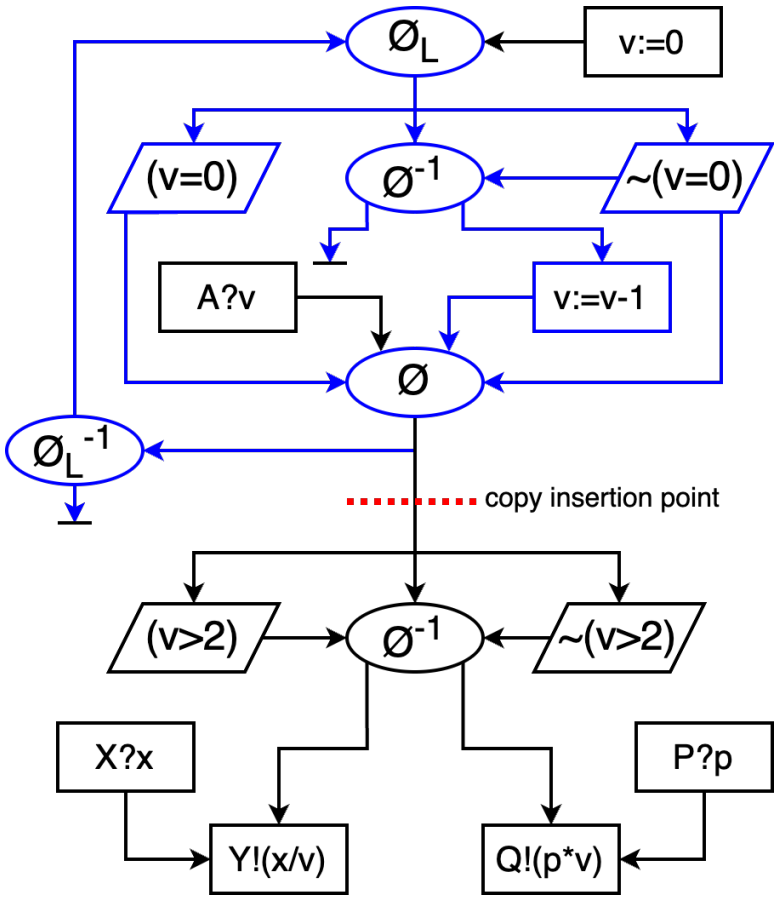


Figure 4.4: DDG for the example CHP program with loop-carried dependencies. Null-terminated edges do not actually exist in the DDG as they do not define a variable, and are shown only for clarity. Nodes in blue are part of the same SCC and hence the blue edges, which are within the same SCC are ignored. 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.

Further, it is not actually modified in the second selection at all. The DDG for this program is shown in Fig. 4.4. Notice that the loop-carried nature of v is captured in the DDG as a strongly-connected component, marked in blue. A strongly-connected component (SCC) in a hypergraph is a maximal set of nodes such that every node is reachable from every other node in the set by traveling along the direction of the directed edge. 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.

Suppose we attempt to break this DDG into two WCCs. We could go about this by cutting two of the hyperedges marked in blue - the one starting at ϕ_L and the one from ϕ to ϕ_L^{-1}. If we do this, then the DDG splits into two WCCs and the resulting CHP would be:

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

This cut has essentially extracted the loop data-dependence out into a pair of channels, i.e. the newly introduced pair C, D. However, this is undesirable as the loop delay now includes two communication actions with no benefit. In general, cutting any hyperedges within an SCC results in this kind of delay increase. Thus, we collapse every SCC in the DDG down into a single supernode and operate on the SCC-graph of the DDG. 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. Since we expect the synthesized circuits to be latency-limited, this is a reasonable constraint to include.

To motivate a better cut choice for this DDG, 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 cutting the hyperedge from the ϕ-node for v, marked in Fig. 4.4. This would result in:

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

4.7 A Simple Heuristic

We have seen how the DDG succinctly captures all data dependencies in the CHP system and, through the process of copy-insertion, allows automatic decomposition. The challenge of finding the optimal decomposition is now reduced to finding a set of cuts maximizes the throughput of the system. However, this is not an easy task as the number of possible cuts is super-exponential in the number and size of the hyperedges. To be precise, there are

$$N = \prod_i B_{|H(e_i)|} \tag{4.1}$$

possible cuts, where B_n is the n-th Bell number that counts the number of ways a set of size n can be partitioned.

In order to tackle this, we provide a simple heuristic to determine which hyperedges to cut. 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 hypergraph, and is implemented using a simple union-find data structure. Following this, there are two passes. The core decision ($|WCC(w')| > 1$) is the same for both passes, and checks whether cutting that particular hyperedge 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 hyperedges would be equivalent to adding a buffer on that channel, and better candidates for decomposition

Algorithm 1: Heuristic Decomposition Algorithm

Input : CHP Program P
Transform selections in CHP into standard form.
Place CHP into static token form.
Build DDG D from CHP.
Compute SCC-hypergraph D_S of D.
$W \leftarrow WCC(D_S)$ ▷ Weakly-connected components
for $w \in W$ **do**
  for $e\ \{n_T, H(e)\} \in w$ **do**
    for $n_H \in H(e)$ **do**
      if $\neg$(*n_T=receive*|*n_H=send)* $\wedge$ $\mathtt{F}\ (w, e, n_T, n_H)$ **then**
        break
      end
    end
  end
  for $e\ \{n_T, H(e)\} \in w$ **do**
    for $n_H \in H(e)$ **do**
      if (*n_T=receive*|*n_H=send)* $\wedge$ $\mathtt{F}\ (w, e, n_T, n_H)$ **then**
        break
      end
    end
  end
end
Rebuild CHP program P from hypergraph D
return: Decomposed CHP Program P

Function $\mathtt{F}\ (w, e, n_T, n_H)$ **:**
  $w' \leftarrow cut(w, e, n_H)$ ▷ cut hyperedge e with $H_x(e) = n_H$
  if $|WCC(w')| > 1$ **then**
    $w \leftarrow w'$
    return **true**
  end
  return **false**
end

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.

4.7.1 Experimental Results

Table 4.2 summarizes the results from performing decomposition on the CHP followed by circuit synthesis with Maelstrom. 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 Table 4.2, 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

Test Program	**CHP Description**	**Synthesis Method**	**Area** (μm^2)	**Latency** (ns)	**Throughput** (MHz)	**Power** (μW)
Sequential Buffers	$*[A?w; W!w;$ $B?x; X!x;$ $C?y; Y!y;$ $D?z; Z!z]$	Sequential	3833	2.84	219	**112**
		Decomposed 1	**3040**	**0.18** (x4)	**2810** (x4)	405
		Decomposed 2	6067	0.35 (x4)	2700 (x4)	793
		Decomposed 3	12125	0.74 (x4)	2670 (x4)	1520
Linear Subprograms	$*[A?x;$ $B?y;$ $C!(x+1), D?z;$ $E!(y+z)]$	Sequential	3258	1.26	271	**138**
		Decomposed 1	**3071**	**0.75, 1.35**	559, 400	304
		Decomposed 2	4897	0.94, 2.11	**563, 408**	442
		Decomposed 3	7807	1.23, 2.84	559, 403	538
Conditional Router	$*[C?c, A?x, B?y;$ $[c=0 \rightarrow X!x$ $[]\, c=1 \rightarrow Y!y]]$	Sequential	**2954**	**1.12**	195	**153**
		Decomposed 1	3849	2.04	437	399
		Decomposed 2	5483	3.41	**444**	629
		Decomposed 3	8282	5.13	442	903
MIPS R3000 Writeback Unit	CHP exactly as in [56]	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 [59]	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

Table 4.2: 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.

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.

4.8 Timing Driven Decomposition

The heuristic algorithm specified earlier does not take into account the characteristics of the circuit that will be generated (i.e. the exact structure of the controller) from the decomposed CHP in order to decide which cuts are good and which are not. The complete, albeit expensive, solution to this problem would be to construct a full optimization loop that involves performing some decomposition, synthesizing the resultant CHP into a circuit, simulating this circuit to determine the cycle period (i.e. throughput) and feeding this information back to the DDG to perform more cuts and so on, until no further improvements can be made or a target throughput is reached. An immediate improvement to the cost of this loop would be to use static timing analysis to extract the cycle time of the synthesized circuit rather than complete device level simulations.

However, synthesizing and timing a circuit still posits the challenge of propagating information back up to the CHP level. In essence, the system needs to keep track of precisely which portions of the circuit are the result of which statements in the CHP and use this mapping to filter hyperedges in the DDG that are considered for copy-insertion. This is impractical for larger designs due to the size of the complete timing graph of the circuit.

Thus, we propose an abstraction: a timing graph model for CHP that is a conservative approximation of the circuit timing graph. Just like the circuit timing graph, the CHP timing graph is also an RER system, and closely resembles the controller structure of the synthesized circuit. This is crucial as the performance of the circuit is determined entirely

by the controller and this control-flow information is complementary to that which the DDG captures, namely data-flow information. Taken together, a complete system that uses all available information in order to decide the best possible decomposition can be constructed.

4.9 CHP Timing Graph

The CHP timing graph is, at its core, a simplified version of the complete timing graph of the entire circuit that the CHP program synthesizes to. In order for this to be well-defined, we must specify which synthesis methodology is being considered. In the rest of this section, we assume that the CHP program is synthesized using the Maelstrom [60] synthesis technique, as this is the current state-of-the-art method for synthesizing asynchronous circuits from arbitrary CHP programs. Further, we also assume a bundled datapath, as this emplaces all the timing information about the circuit entirely within the control path. All the delay information of the datapath is encoded in the matched delay lines that are placed in the control path.

With knowledge of the relationship between a given CHP program and the synthesized circuit, we can derive an RER timing model for the CHP. In order to see how this works, consider the following example CHP:

$$*[L?x;\ \ y := f(x);\ \ R!g(y)]$$

Fig. 4.5 shows the control circuit generated by Maelstrom for this CHP. The circuit is a ring, with the right edge looping back and connecting to the left edge. To keep matters simple, we consider one phase of operation of this loop. Initially, all gate outputs reset to low and upon coming out of reset, the first gate to fire is the left-most C-element, which is essentially an initial token buffer that controls the ring, activating the ring. Hence, this marks the beginning of the iteration and we have a single node with a ticked edge

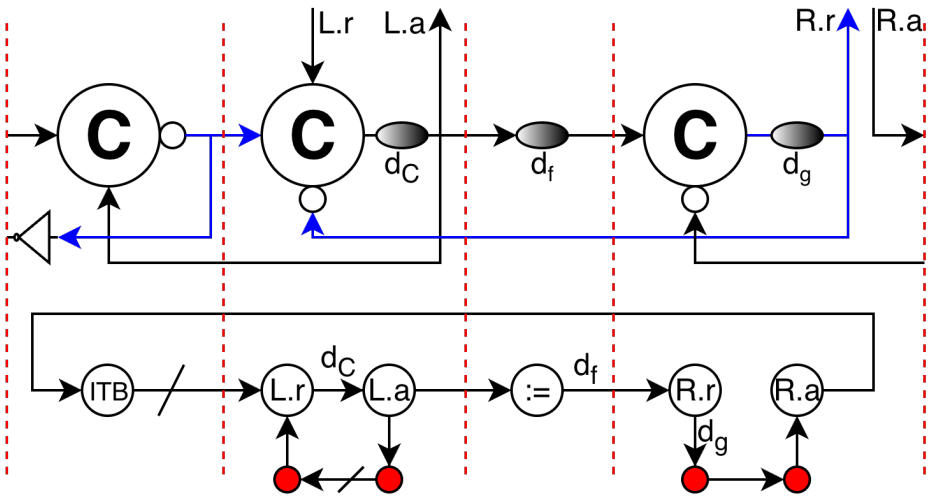


Figure 4.5: Top: Control Circuit resulting from synthesis of the example CHP. The right edge is folded back to the left edge, making the two connections that close the loop. Bottom: Corresponding CHP timing graph. Red nodes denote terminating source/sink nodes added to create a closed graph.

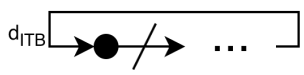
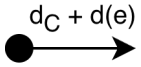
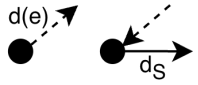
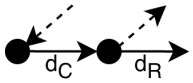
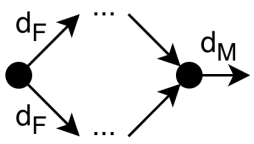
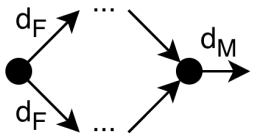
emanating from it. Next, the activated second C-element ($L?x$) fires whenever the channel request $L.r$ is raised and, after a delay (d_C) corresponding to the time taken for the datapath to capture the value (the interface between the control and datapath where pulse generators are triggered is omitted for clarity), activates the following circuit. We can capture this behavior by having a request node that depends on the two aforementioned events, with an outgoing edge of weight d_C. The acknowledge node connects to the environment and the following circuit. The assignment ($y := f(x)$) is trivial at it is simply a delay d_f corresponding to the delay of the combinational logic that implements the expression. We simply have one node with the appropriate delay for this. Next, the send ($R!g(y)$) has a delay d_g for calculating the function and then sends out the request to the environment. The downstream portion of the ring is activated by the acknowledge from the environment. We can model this once again by a pair of request and acknowledge nodes as shown, but without an edge connecting them, as the causal influence must come from the environment. Finally, the connection from the end of the ring loops back to the ITB, closing this loop in the timing graph.

This timing graph essentially captures one half-phase of the circuit loop iteration. It is straightforward to see that the corresponding next half-phase will have the exact same sequence of events, but with all the senses flipped (i.e. rising edges replaced with falling edges and vice versa). Finally, notice that the timing graph described so far contains dangling edges - this is due to the fact that we do not have a closed system in CHP. In order to close the graph, we assume that all primary input/output channels are terminated with sources/sinks respectively. Effectively, this creates the red nodes in the timing graph in Fig. 4.5, which are essentially an infinitely fast source $*[L!]$ and sink $*[R?]$ that model the fastest possible environment.

Table 4.3 shows the exact correspondence between CHP constructs and the timing graph. Each assignment corresponds to a single node. A channel action corresponds to two nodes, one for the channel request and one for the acknowledge. Selections and parallel constructs correspond to fork and join nodes. The edges encode the ordering of these actions and their weights correspond to the delay of performing a particular action, i.e. assignment, send or receive. The only source of ticked edges is the loop construct, which essentially denotes the boundary between one iteration of the loop and the next. The delays corresponding to each edge are also marked. An unmarked edge implies that its delay is zero. The dotted edges simply represent ones that cross between processes but are not different in any way.

The delays in the timing graph are derived in one of two ways. For the controller delays, we extract them from SPICE simulations, which can be done once per technology and then stored in a lookup table as they do not depend on the written CHP. The only delay that is not available statically is the delay of an expression, as the CHP can contain an arbitrary number of arbitrary kinds. To get this delay, we call a conventional logic synthesis tool such as ABC [44] or Yosys [46] and extract the delay from the generated combinational circuit. The timing graph constructed in this way allows us to closely approximate the timing behavior of the post-synthesis circuit without actually having to construct the

Table 4.3: CHP Timing Graph Sub-components

CHP	TG Component
$*[...]$	
$x := e$	
$C!e$	
$C?x$	
$[g_1 \rightarrow P_1 [] g_2 \rightarrow P_2]$	
P_1, P_2	

d_{ITB} is the delay of the initial token buffer that initializes the ring. d_C is the capture delay of the data-storage element. $d(e)$ is the delay of the combinational logic for an expression. d_S and d_R are the delays of the send and receive controller elements. d_F and d_M are the delays of the selection/parallel fork and merge elements. These have different values based on whether it is a selection or parallel, but the structure is identical.

full detailed timing graph. Once the timing graph has been constructed, we extract the critical cycle using an implementation of the Young-Tarjan-Orlin (YTO) algorithm [61].

4.10 Using the Timing Information

The critical cycle on the timing graph tells us the path that limits the throughput of the system. Note that this is always a conservative approximation. For example, if the critical path through the program contained the following CHP fragment:

$$[c \longrightarrow fast_1 [] \neg c \longrightarrow slow_1];$$
$$[c \longrightarrow slow_2 [] \neg c \longrightarrow fast_2]$$

then it is evident that the delays of $slow_1$ and $slow_2$ will never add up on the same iteration, but discerning this requires knowledge of the values of variables at runtime, something that requires simulation of the program in general and cannot be statically determined from the structure alone. In such cases, the timing graph would be an overestimate and would have the sum of the delays of the two slow paths in the critical path.

Once we determine the nodes on the critical cycle in the timing graph, we make this information actionable by determining which nodes in the DDG these timing nodes correspond to. The correspondences for the assignment, send and receive nodes are straightforward, they simply map to their respective assignment, send and receive nodes in the DDG. The parallel fork and merge nodes do not map to any node in the DDG. The selection fork and merge nodes map to the guard, ϕ and ϕ^{-1} nodes of that particular selection. The loop nodes map to the ϕ_L and ϕ_L^{-1} nodes of that particular loop.

After performing this mapping, we arrive at a set of nodes in the DDG upon which to focus efforts of performing hyperedge cuts. Recall from earlier that in order to avoid exacerbating data-dependence loops, we collapse strongly-connected components (SCCs) in the DDG down into a single super-node and operate on this SCC-graph. Trying every possible hyperedge associated with this set of nodes is still expensive and thus we propose the following technique. Consider some topologically sorted list L of the set of nodes under consideration. For every prefix list L_P of L, cut all the hyperedges with:

1. heads in L_P and tails not in L_P

2. tails in L_P and heads not in L_P

Effectively, this extracts the nodes in L_P out into a component that is disconnected from the main graph. Since we know that each component of the graph corresponds to a single

Algorithm 2: Hyperedge Choice Algorithm

Input : DDG SCC-Graph D, Timing Graph T
$L_H \leftarrow []$
$N_{CC} \leftarrow Nodes(Crit(T))$ ▷ Nodes on critical cycle
$N_D \leftarrow Map(N_{CC}, D)$ ▷ Map to D nodes
$LL_P \leftarrow PrefixLists(Toposort(N_D))$
for $L_P \in LL_P$ **do**
  $H \leftarrow \{\}$
  for $h \in D$ **do**
    if $(Head(h) \in L_P \wedge Tail(h) \notin L_P)$
     $\vee(Head(h) \notin L_P \wedge Tail(h) \in L_P)$ **then**
      $H.add(h)$
    end
  end
  $L_H.append(H)$
end
return: List of Set of Hyperedges L_H

process, the part of the critical cycle that was in L_P has now been extracted to a separate process, and we are guaranteed that the critical cycle is broken. The exact algorithm for performing the hyperedge choice is shown in Algorithm 2.

Now that we have an effective method of narrowing down the search space based on the timing graph, we turn to the outer optimization loop. The top-level decomposition algorithm (Algorithm 3) works by iteratively constructing the DDG and the timing graph, trying copy-insertion on a particular set of hyperedges and checking for an improvement. Note that it is not possible to incrementally update the timing graph based on the copy insertion as the change in structure of the data dependencies might change the resulting CHP in non-trivial ways. Thus, the construction of the new program P_{test} is required. The algorithm iteratively performs these steps until the cycle time target specified by the user is reached or no further improvement is possible.

Algorithm 3: Timing-Driven Decomposition Algorithm

Input : CHP Program P, Cycle Time Target C_{target}
do
  $D \leftarrow DDG(P)$
  $D_{SCC} \leftarrow SCC(D)$
  $T \leftarrow TG(P)$ ▷ Timing Graph
  $L_H \leftarrow HyperedgeChoice(D_{SCC}, T)$ ▷ Algorithm 2
  $h_{best} \leftarrow \{\}, C_{best} \leftarrow Crit(T).$
  for $h_{set} \in L_H$ **do**
    $P_{test} \leftarrow Insert\ copies\ (P, h_{set})$
    $D' \leftarrow DDG(P_{test})$
    if $WCC(D') > WCC(D)$ **then**
      $T' \leftarrow TG(P_{test})$
      if $Crit(T') < C_{best}$ **then**
        $h_{best} \leftarrow h_{set}$
        $C_{best} \leftarrow Crit(T')$
      end
    end
  end
  $P \leftarrow Insert\ copies\ (P, h_{best})$
while $(Crit(T)\ improved\ \wedge\ Crit(T) > C_{target})$;
return: Decomposed CHP Program P

4.11 Automatic Slack Matching

Our proposed framework allows detection of the portion of the program that is the bottleneck for throughput via identifying the nodes that are on the critical cycle of the timing graph. However, nothing restricts these nodes to all lie within the same process. Since the timing graph also contains connectivity across channels, it is possible that the critical cycle threads through several processes. Note that this is unlike the DDG, where channel dependencies are not explicitly present as hyperedges, i.e. there is no hyperedge from a $C!e$ node to a $C?x$ node for the same channel C. Fig. 4.6 (top) shows the timing graph (with the nodes on the critical cycle marked) for the following CHP, which corresponds to a simple unbalanced branched pipeline:

$$*[A!v_1, C!v_2] \parallel *[A?v_3; B!v_3] \parallel *[B?v_4, C?v_5]$$

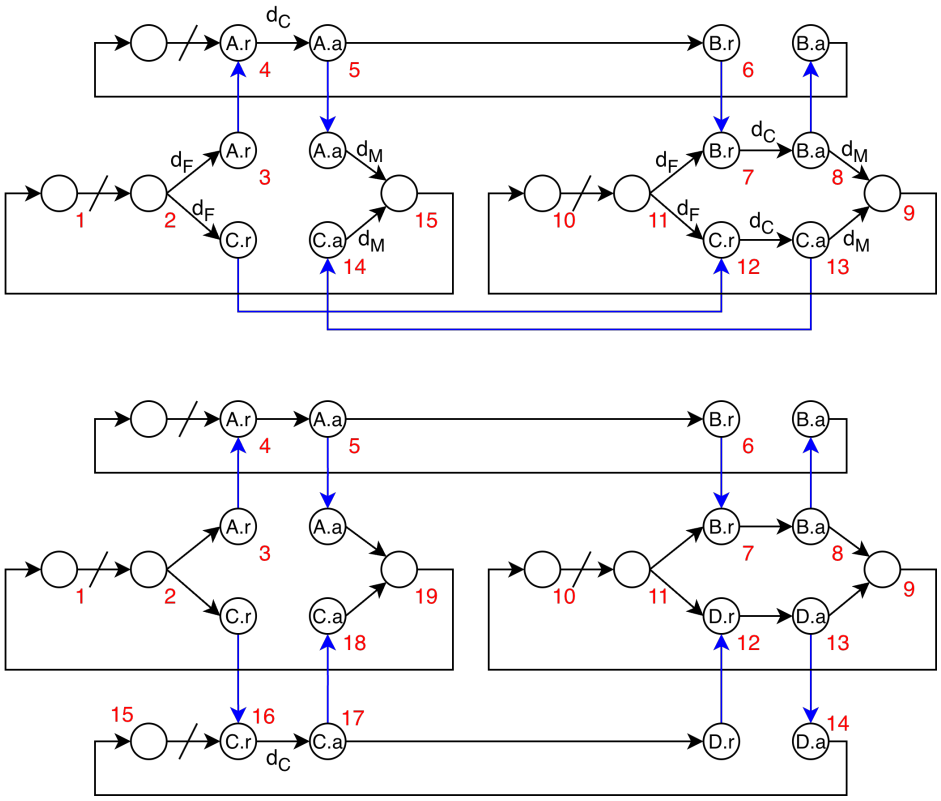


Figure 4.6: Timing Graph with multiple processes. Nodes marked in red show the critical cycle. Top: before copy-insertion. Bottom: after copy-insertion on channel C.

In this case, the critical cycle has two ticks on it and threads through all the processes. Recall from earlier that performing copy insertion is essentially introducing a channel communication to move the data dependence from a variable to this channel. A trivial case of this is inserting a channel on the variable of a buffer. In this case, some of the nodes on the critical cycle correspond to the CHP that contains the variable v_2, and performing copy-insertion on these is essentially adding a buffer on channel C. If this is done, then we have in effect balanced the pipeline automatically. The critical cycle on the new timing graph, shown in Fig. 4.6 (bottom), will have an added tick on it that will offset the cost of having added weight from the latency of inserting the buffer on C. The cycle mean of this

particular cycle gets modified as:

$$p_{new}(C) = \frac{\sum d(e_i) + d_C}{\sum \epsilon(e_i) + 1}, \;\; p_{old}(C) = \frac{\sum d(e_i)}{\sum \epsilon(e_i)} \tag{4.2}$$

The delay cost of adding a buffer is simply one capture delay (d_C), but this provides the benefit of increasing the denominator by 1, which outweighs the cost. It is straightforward to check that $p_{new} < p_{old}$ whenever $d_C < p_{old}$, which is satisfied for all systems of interest. If this is not true, then it would imply that the critical period of the system is already smaller than the critical period of a single buffer (discounting even the controller overhead), which is only true for trivial systems such as $*[skip]$. The CHP for the slack-matched system is:

$$*[A!v_1, C!v_2] \parallel *[A?v_3; B!v_3] \parallel *[B?v_4, D?v_5] \parallel *[C?v_6; D!v_6]$$

As a more interesting example, consider this CHP system:

$$*[U0!a, D0!b] \;\; (1)$$
$$\parallel *[U0?x1; U1!x1, M1!x1] \;\; (2)$$
$$\parallel *[U1?x2, M0?x3; U2!x2 \wedge x3] \;\; (3)$$
$$\parallel *[U2?x4; U3!x4] \;\; (4)$$
$$\parallel *[D0?x5; M0!x5, D1!x5] \;\; (5)$$
$$\parallel *[U3?z1, M1?z2, D1?z3] \;\; (6)$$

This is a pipeline with cross-linked branches (block diagram shown in Fig. 4.7) where the number of stages and hence the delay on the different possible paths through the pipeline are significantly different from each other. The critical cycle is usually a multi-process cycle like before, with several ticks on it. In general, determining the optimal locations and number of buffers to insert is a non-trivial task. However, using the CHP timing graph, the decomposition system is able to automatically determine the best slack

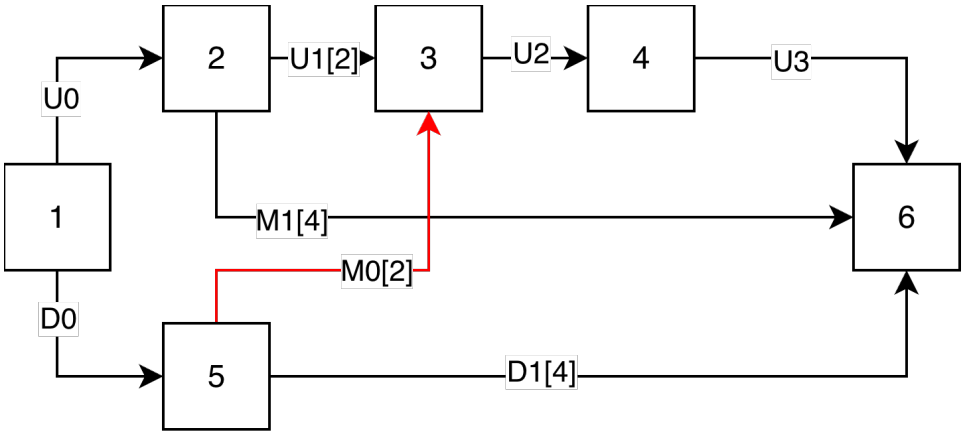


Figure 4.7: Block diagram of example CHP system. Edges represent channels. The numbers on the edge represent the optimal amount of buffering to add on that particular channel in order to match slack on the different branches of the pipeline.

addition strategy, i.e. that of adding slack $(2, 4, 2, 4)$ on channels $(U1, M1, M0, D1)$ respectively. Note that any slack addition on these channels of the form $(n, n+2, n, n+2)$ is a valid solution at first glance. However, as n increases, there is one buffer's worth of delay added to the numerator, and one unit added to the denominator (see Equation 4.2). The optimal choice of n thus depends on the exact delays in the system is thus a technology-dependent value. The choice of $n = 2$ that the system makes is a result of the technology under consideration.

In essence, we have unified the process of process decomposition and slack-matching as they are really the same challenge - that of minimizing the critical cycle of the system of processes.

4.12 Expression Pipelining

So far, we have considered optimization methods on CHP that operate at a level where each statement (send, receive or assignment) is an atomic construct. However, there are cases where the cycle time target can be limited by a single expression. For example, consider: $*[L_1?x, L_2?y; R!(x*y)]$. Here, no amount of DDG-based decomposition can

reduce the cycle time to below that of the delay of the combinational logic corresponding to the multiply operation. In these cases, we need to fragment the computation of the expression $(x{*}y)$ into multiple stages. Suppose we could find a functions $f_1(\cdot,\cdot), f_2(\cdot), f_3(\cdot)$ such that $f_3(f_2(f_1(x,y))) \equiv x * y$. Then we could rewrite the original program as: $*[L_1?x, L_2?y; v_1 := f_1(x,y); v_2 := f_2(v_1); R!f_3(v_2)]$, which would allow the decomposition to break this apart into:

$$*[L_1?x, L_2?y; C_1!f_1(x,y)] \parallel *[C_1?t_1; C_2!f_2(t_1)] \parallel *[C_2?t_2; R!f_3(t_2)]$$

resulting in the cycle time being limited by the maximum of the delay of the three functions. A trivial solution would be for two of the functions to be the identity, and one being the entire multiply operation. However, this evidently leads to no benefit, and coming up with the optimal decomposition without gate-level information of the logic implementing the expression is not straightforward.

Conventionally, for combinational logic, the equivalent operation is done at the netlist level via retiming. However, we would like to achieve the same at the CHP level. In order to implement this, we simply synthesize the combinational logic for the expression using a standard logic synthesis tool such as ABC [62], and use built-in commands to place and move latches to the appropriate locations within the logic network. This effectively breaks up the entire combinational block into n stages. We then convert the logic equations corresponding to each combinational sub-block to construct the requisite functions $f_i(\cdot)$, effectively achieving our goal. Note that this is timing-aware as ABC uses information about the delays of each gate in the cell library to decide final locations for the latches, and thus the exact nature of each sub-block. Hence, this operation, just like the others, is technology-dependent.

We incorporate this expression pipelining/fragmentation as a pre-processing step where each expression in the CHP that exceeds the cycle time target is broken up sufficiently such that the delay of each of the pieces is individually smaller than the cycle time target by

some margin. This margin, which is typically small, is required as the actions of receiving and sending have some delay, which arise from the control circuit and the capture delay of the data-storage element, as can be seen by the sub-processes in the example above. The delay of $*[C_1?t_1; C_2!f_2(t_1)]$ consists of delays beyond just that of f_2.

4.13 Results

Table 4.4 summarizes the results from performing timing-driven decomposition on the CHP followed by circuit synthesis with the optimized implementation of Maelstrom. 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. In contrast to the heuristic-based decomposition technique, there is no user-level iteration needed in order to decide the best possible decomposition. The timing-driven decomposition system decides the stopping point automatically, as described in Algorithm 3.

As before, decomposition typically results in higher throughput at the cost of a larger circuit, increased latency and power consumption. This is the result of added communications that are inserted while performing copy-insertions. For the multiplier test case, the resulting cycle time (inverse of throughput) is smaller than the delay of the single-stage consisting of just a multiplier. This level of optimization is enabled by the expression pipelining step. A similar argument applies for the simple ALU test case as well. For the cross-linked pipeline, each pipeline stage has a relatively small delay and as such, the critical cycle is usually the multi-process cycle and adding a buffer on the channels in this cycle result in an added tick, reducing the critical period. The timing-driven decomposition repeats this until the critical cycle is contained within a single process and thus adding a buffer would have no effect.

Currently, the expression pipelining step has the potential drawback of fragmenting

Test Program	**CHP Description**	**Synthesis Method**	**Area** (μm^2)	**Latency** (ns)	**Throughput** (MHz)	**Power** (μW)
Linear Subprograms	$*[A?x; B?y;$ $C!(x+1), D?z; E!(y+z)]$	Sequential	2409	1.24	810	321
		Decomposed	6035	3.61	1460	1120
Conditional Router	$*[C?c, A?x, B?y;$ $[c=0 \rightarrow X!x [] c=1 \rightarrow Y!y]]$	Sequential	2905	1.01	976	315
		Decomposed	6115	3.13	1227	509
Multiplier	$*[L_1?x_1, L_2?x_2$ $R!(x_1 * x_2)]$	Sequential	2732	1.63	595	140
		Decomposed	6838	2.78	1242	780
Simple ALU	$*[I1?x_1, I2?x_2, C?op;$ $y := f(x_1, x_2, op); O!y]$	Sequential	6098	1.87	525	130
		Decomposed	11426	3.32	676	430
MIPS R3000 Writeback Unit	CHP exactly as in [56]	Sequential	7915	1.42	719	212
		Decomposed	15213	3.95	1653	1230
Async. RISC-V Fetch Unit	Microarchitecture detailed in [59]	Sequential	46456	3.16	315	945
		Decomposed	68098	5.74	671	2670
Cross-linked Pipeline (Slack-matching)	See Fig. 4.7	Sequential	5458	4.45	1015	634
		Decomposed	9428	6.24	1235	1340

Table 4.4: Synthesis results for CHP programs using the optimized implementation of Maelstrom, with and without timing-driven decomposition. All metrics are from SPICE simulations in a 65nm technology node.

a single assignment in such a way that the intermediate assignments might have a large bitwidth. When copy insertion is performed on these nodes, this results in a fresh channel that has an equally large bitwidth, which results in an expensive circuit. This artifact can be observed in the ALU test case where the power consumption more than triples but the throughput gain is not significant. This power expenditure is the result of the function $f(x_1, x_2, op)$ being split at a point that is too wide.

The decomposition technique based on the timing graph and the DDG currently only try to maximize the throughput of the system without regard for the circuit cost of the added communication actions. In the future, it would be beneficial to incorporate this cost into the decomposition algorithm. Finally, orthogonal to this, there has been work on sharing expensive hardware resources without sacrificing performance [63, 64]. 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.

4.14 Discussion

The iterative optimization loop described in Algorithm 3 requires several calculations of the critical cycle on the timing graph. The YTO algorithm that is used for this has a worst-case time complexity of $O(V \cdot E + V^2 \cdot log(V))$, where V is the number of the vertices in the graph and E is the number of edges. However, experiments on graphs with $(V + E) \approx 10^5$ have shown that the practical time complexity of YTO is $O(V \cdot log(V))$, which is much smaller [65]. In our experiments as well, we observe that the critical cycle calculation forms a negligible portion of the runtime, which is dominated by the combinational logic synthesis tools.

The core optimization technique presented here works by splintering the original CHP program into several based on data dependencies. There has been similar work [37, 66,

67], that relies on composing canopy graphs of several pipeline components and matching the dynamic slack in order to achieve higher throughput. However, the usefulness of this technique is restricted by the fact that channel accesses may occur at arbitrary points in a given system of CHP programs. This means that even if the original system can be represented as a properly nested pipeline, this is not true once decompositions are applied. The technique presented here does not rely on such assumptions and can handle an arbitrary system of CHP programs.

4.15 Summary

In this chapter, we present 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. The decomposition technique relies on a novel timing model of the circuit that can be derived directly from the CHP, and uses this to focus optimization efforts on the parts of the program that are the limiting factors for performance. 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.

Chapter 5

Future Work

5.1 Control Circuit Families

The circuit synthesis techniques described earlier are agnostic to the family of control circuits that are used. Throughout this work, we used QDI C-element control circuits to expound the synthesis approach. We also provided brief summaries of how this can be extended to the MOUSETRAP and GasP families. By deriving the necessary control elements (i.e. Send, Receive, Selection, Parallel) for a given family, support for these can be added as well.

5.2 Process Technologies

The synthesis and decomposition techniques currently supports a commercial 65nm manufacturing technology. However, support for newer technologies can be added by performing a suite of SPICE simulations and characterizations. These provide information about the delays of various circuit elements that allow the synthesis to instantiate precise delay lines. Further, the timing-driven decomposition can also use this information to arrive at a decomposed system that is optimal for a given technology.

5.3 Physical Optimizations

The circuit synthesis procedure currently does not take physical constraints into account. When the generated circuit is placed and routed, the delays along different paths in the circuit may change and this might necessitate tuning of delay lines and insertion of delay buffers in order to ensure that timing constraints are satisfied. Further, the generated circuits will require sizing of the transistors that depends on the load that each gate needs to drive. This needs to be done at the netlist level by analyzing the connectivity of the entire circuit and is required in order to achieve the maximum performance from any given controller topology.

5.4 Template Recognition

The synthesis and optimization techniques described in this work preserve the precise behavior of the program. However, it might be possible to perform higher-level optimizations by relaxing this constraint slightly. These optimizations can be realized in the form of CHP rewrites that preserve the externally observable behavior of a particular process while significantly changing the internals, and/or a library of optimized circuits that are used to implement frequently used processes.

As an example, consider an accumulator process such as the one below:

$$s := 0; *[C?c; X?x; [c = 1 \longrightarrow s := 0; S!s [] c = 0 \longrightarrow s := s + x; S!s]]$$

This seems like a perfectly valid implementation. However, this contains unnecessary selections and multiple channel accesses that can limit the performance of the overall system. A better implementation of the same would be:

$$s := 0; \;\; *[C?c, X?x; s := ((c = 1)?0 : s + x); S!s]$$

This degree of structural change to the written CHP is currently beyond the capabilities of the optimization system. Further, notice that the program is now much simpler and thus, it is possible to come up with an optimized circuit for this (similar to the procedure described in Section 3.14). Exploring these two steps further and implementing them form important avenues for future exploration.

Bibliography

[1] M. J. S. Smith, *Application-specific integrated circuits*, vol. 7. Addison-Wesley Boston, 1997. 1

[2] J. Shinde and S. Salankar, "Clock gating—a power optimizing technique for vlsi circuits," in *2011 annual IEEE India conference*, pp. 1–4, IEEE, 2011. 1

[3] Q. Wu, M. Pedram, and X. Wu, "Clock-gating and its application to low power design of sequential circuits," *IEEE transactions on circuits and systems I: fundamental theory and applications*, vol. 47, no. 3, pp. 415–420, 2000. 1

[4] J. Muttersbach, T. Villiger, and W. Fichtner, "Practical design of globally-asynchronous locally-synchronous systems," in *Proceedings Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2000)(Cat. No. PR00586)*, pp. 52–59, IEEE, 2000. 2

[5] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems.," tech. rep., 1984. 2

[6] J. Sparsø, *Introduction to asynchronous circuit design*. Kongens Lyngby, Denmark: DTU Compute, Technical University of Denmark, 2020. OCLC: 1199326564. 2

[7] S. M. Nowick and M. Singh, "Asynchronous Design—Part 1: Overview and Recent Advances," *IEEE Design & Test*, vol. 32, pp. 5–18, June 2015.

[8] S. M. Nowick and M. Singh, "Asynchronous Design—Part 2: Systems and Methodologies," *IEEE Design & Test*, vol. 32, pp. 19–28, June 2015. 2

[9] D. Edwards and A. Bardsley, "Balsa: An asynchronous hardware synthesis langugae," *The Computer Journal*, vol. 45, no. 1, pp. 12–18, 2000. 2, 19, 58

[10] S. F. Nielsen, J. Sparsø, J. B. Jensen, and J. S. R. Nielsen, "A behavioral synthesis frontend to the haste/tide design flow," in *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, pp. 185–194, IEEE, 2009.

[11] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An asic flow for ghz asynchronous designs," *IEEE Design & Test of Computers*, vol. 28, no. 5, pp. 36–51, 2011.

[12] I. Poliakov, V. Khomenko, and A. Yakovlev, "Workcraft–a framework for interpreted graph models," in *International Conference on Applications and Theory of Petri Nets*, pp. 333–342, Springer, 2009. 2

[13] S. Ataei, W. Hua, Y. Yang, R. Manohar, Y.-S. Lu, J. He, S. Maleki, and K. Pingali, "An open-source eda flow for asynchronous logic," *IEEE Design & Test*, vol. 38, no. 2, pp. 27–37, 2021. 2

[14] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978. 3

[15] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, p. 720–738, jun 1989. 7, 21, 57

[16] R. Manohar, T.-K. Lee, and A. J. Martin, "Projection: A synthesis technique for concurrent systems," in *Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 125–134, IEEE, 1999. 11, 79

[17] 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*, pp. 508–513, 2003. 11, 20

[18] 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.), vol. 1422, (Berlin, Heidelberg), pp. 272–285, Springer Berlin Heidelberg, 1998. Series Title: Lecture Notes in Computer Science. 11, 70, 72

[19] 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), pp. 17–27, IEEE, 2004. 11, 70, 80

[20] C. S. Ananian, *The static single information form*. PhD thesis, Massachusetts Institute of Technology, 2001. 12

[21] S. M. Burns and A. J. Martin, "Performance Analysis and Optimization of Asynchronous Circuits," in *Conference on Advanced Research in VLSI*, 1990. 14

[22] W. Hua, Y.-S. Lu, K. Pingali, and R. Manohar, "Cyclone: A Static Timing and Power Engine for Asynchronous Circuits," in *2020 26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, (Salt Lake City, UT, USA), pp. 11–19, IEEE, May 2020. 14, 16

[23] S. H. Unger, "Hazards and delays in asynchronous sequential switching circuits," *IRE Transactions on Circuit Theory*, vol. 6, no. 1, pp. 12–25, 1959. 18

[24] S. M. Nowick and D. L. Dill, "Automatic synthesis of locally-clocked asynchronous state machines," in *IEEE International Conference on Computer-Aided Design*, pp. 318–319, IEEE Computer Society, 1991. 18

[25] R. Manohar and Y. Moses, "The eventual c-element theorem for delay-insensitive asynchronous circuits," in *IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 102–109, 2017. 19

[26] A. J. Martin, "Synthesis of asynchronous vlsi circuits," Tech. Rep. CS-TR-93-28, California Institute of Technology, 1991. 19

[27] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, vol. 80, no. 3, pp. 315–325, 1997. 19

[28] R. Manohar, "An analysis of reshuffled handshaking expansions," in *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001*, pp. 96–105, IEEE, 2001. 19

[29] A. M. Lines, "Pipelined asynchronous circuits," Master's thesis, California Institute of Technology, 1995. 19

[30] R. Manohar and A. J. Martin, "Slack elasticity in concurrent computing," in *Mathematics of Program Construction* (J. Jeuring, ed.), (Berlin, Heidelberg), pp. 272–285, Springer Berlin Heidelberg, 1998. 19, 20

[31] R. Manohar, "chp2prs: Syntax-directed translation of chp programs into production rules." `https://github.com/asyncvlsi/chp2prs`, 2023. 19, 68

[32] S. M. Burns and A. J. Martin, "Syntax-directed translation of concurrent programs into self-timed circuits," in *Conference on Advanced Research in VLSI*, 1988. 19, 59

[33] K. van Berkel and M. Rem, "Vlsi programming of asynchronous circuits for low power," in *Asynchronous Digital Circuit Design*, pp. 151–210, Springer, 1995. 19

[34] H. S. Inc. `https://web.archive.org/web/20090323054030/http://www.handshakesolutions.com/`, 2009. 19

[35] 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)*, pp. 1–8, IEEE, 2021. 20, 61

[36] S. Taylor, D. Edwards, and L. Plana, "Automatic compilation of data-driven circuits," in *2008 14th IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 3–14, IEEE, 2008.

[37] 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*, pp. 107–116, IEEE, 2010. 20, 101

[38] A. J. Martin, S. M. Burns, T.-K. Lee, D. Borkovic, and P. J. Hazewindus, "The design of an asynchronous microprocessor," *SIGARCH Comput. Archit. News*, vol. 17, no. 4, pp. 99–110, 1989. 21

[39] A. J. Martin, "Asynchronous datapaths and the design of an asynchronous adder," *Formal Methods in System Design*, vol. 1, pp. 117–137, 1992. 26

[40] A. V. Aho, M. S. Lam, and J. D. Ullman, *Compilers: Principles, Techniques & Tools*. London, England: Pearson Education, 2007. 29

[41] M. Nyström, R. Manohar, and A. J. Martin, "Method and apparatus for a failure-free synchronizer," Feb. 10 2004. US Patent 6,690,203. 34

[42] S. Tugsinavisut, Y. Hong, D. Kim, K. Kim, and P. Beerel, "Efficient asynchronous bundled-data pipelines for dct matrix-vector multiplication," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 4, pp. 448–461, 2005. 54

[43] R. Tadros, N. Dasari, and P. Beerel, "Ultra-low power pass-transistor-logic-based delay line design for sub-threshold applications," *Electronics Letters*, vol. 52, no. 23, pp. 1910–1912, 2016. 54

[44] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference*, (Germany), pp. 24–40, Springer, Berlin, 2010. 58, 90

[45] A. Mishchenko, S. Chatterjee, and R. Brayton, "Dag-aware aig rewriting a fresh look at combinational logic synthesis," in *Proceedings of the 43rd annual Design Automation Conference*, pp. 532–535, 2006. 58

[46] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, vol. 97, 2013. 58, 90

[47] W. Hua, Y.-S. Lu, K. Pingali, and R. Manohar, "Cyclone: A static timing and power engine for asynchronous circuits," in *2020 26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 11–19, IEEE, 2020. 58

[48] S. M. Burns and A. J. Martin, "Performance analysis and optimization of asynchronous circuits," in *Conference on Advanced Research in VLSI*, 1990.

[49] W. Hua and R. Manohar, "Exact timing analysis for asynchronous systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 203–216, 2017. 58

[50] J. Teifel and R. Manohar, "Static tokens: Using dataflow to automate concurrent pipeline synthesis," in *10th International Symposium on Asynchronous Circuits and Systems*, pp. 17–27, IEEE, 2004. 61

[51] M. Singh and S. M. Nowick, "Mousetrap: High-speed transition-signaling asynchronous pipelines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 684–698, 2007. 66

[52] I. Sutherland and S. Fairbanks, "Gasp: A minimal fifo control," in *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems*, pp. 46–53, IEEE, 2001. 66

[53] R. Manohar, "chp2prs: Syntax-directed translation of CHP programs into production rules," 2023. 70

[54] A. Bardsley and D. Edwards, "The balsa asynchronous circuit synthesis system," in *Forum on Design Languages*, vol. 224, 2000. 70

[55] 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), pp. 1–8, IEEE, Sept. 2021. 70, 80

[56] 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), pp. 125–134, IEEE Comput. Soc, 1999. 70, 86, 100

[57] 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), pp. 508–513, ACM, June 2003. 70

[58] A. V. Aho, M. S. Lam, and J. D. Ullman, *Compilers: Principles, Techniques & Tools*. London, England: Pearson Education, 2007. 71

[59] R. Dashkin, *Asynchronous RISC-V CPU Design With Pre-silicon Validation on Synchronous FPGAs*. PhD thesis, Yale University, 2024. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2024-08-11. 86, 100

[60] K. Srinivasan and R. Manohar, "Maelstrom: A logic synthesis technique for asynchronous circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025. 88

[61] N. E. Young, R. E. Tarjant, and J. B. Orlin, "Faster parametric shortest path and minimum-balance algorithms," *Networks*, vol. 21, no. 2, pp. 205–221, 1991. 91

[62] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *Computer Aided Verification* (D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, T. Touili,

B. Cook, and P. Jackson, eds.), vol. 6174, (Berlin, Heidelberg), pp. 24–40, Springer Berlin Heidelberg, 2010. Series Title: Lecture Notes in Computer Science. 98

[63] 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*, pp. 107–116, 2010. 101

[64] 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*, pp. 57–64, 2012. 101

[65] A. Dasdan, "Experimental analysis of the fastest optimum cycle ratio and mean algorithms," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 9, no. 4, pp. 385–418, 2004. 101

[66] A. M. Lines, *Pipelined Asynchronous Circuits*. PhD thesis, 1998. Publisher: California Institute of Technology. 101

[67] J. Hansen and M. Singh, "An energy and power-aware approach to high-level synthesis of asynchronous systems," in *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 269–276, IEEE, 2010. 102