Pipelined Asynchronous High Level Synthesis for General Programs

Rui Li

Yale University Graduate School of Arts and Sciences, rui.lee92@gmail.com

Follow this and additional works at: https://elischolar.library.yale.edu/gsas_dissertations

Recommended Citation
Li, Rui, "Pipelined Asynchronous High Level Synthesis for General Programs" (2021). Yale Graduate School of Arts and Sciences Dissertations. 367.
https://elischolar.library.yale.edu/gsas_dissertations/367

This Dissertation is brought to you for free and open access by EliScholar – A Digital Platform for Scholarly Publishing at Yale. It has been accepted for inclusion in Yale Graduate School of Arts and Sciences Dissertations by an authorized administrator of EliScholar – A Digital Platform for Scholarly Publishing at Yale. For more information, please contact elischolar@yale.edu.
Abstract

Pipelined Asynchronous High Level Synthesis for General Programs

Rui Li
2021

High-level synthesis (HLS) translates algorithms from software programming language into hardware. We use the dataflow HLS methodology to translate programs into asynchronous circuits by implementing programs using asynchronous dataflow elements as hardware building blocks. We extend the prior work in dataflow synthesis in the following aspects: i) we propose Fluid to synthesize pipelined dataflow circuits for real-world programs with complex control flows, which are not supported in the previous work; ii) we propose PipeLink to permit pipelined access to shared resources in the dataflow circuit. Dataflow circuit results in distributed control and an implicitly pipelined implementation. However, resource sharing in the presence of pipelining is challenging in this context due to the absence of a global scheduler. Traditional solutions to this problem impose restrictions on pipelining to guarantee mutually exclusive access to the shared resource, but PipeLink removes such restrictions and can generate pipelined asynchronous dataflow circuits for shared function calls, pipelined memory accesses and function pointers; iii) we apply several dataflow optimizations to improve the quality of the synthesized dataflow circuits; iv) we implement our system (Fluid + PipeLink) on the LLVM compiler framework, which allows us to take advantage of the optimization efforts from the compiler community; v) we compare our system with a widely-used academic HLS tool and two commercial HLS tools. Compared to commercial (academic) HLS tools, our system results in 12X (20X) reduction in energy, 1.29X (1.64X) improvement in throughput, 1.27X (1.61X) improvement in latency at a cost of 2.4X (1.61X) increase in the area.
Pipelined Asynchronous High Level Synthesis for General Programs

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

by
Rui Li

Dissertation Director: Rajit Manohar

December 2021
Contents

1 Introduction .......................... 1
   1.1 Motivation .......................... 1
   1.2 HLS Overview ......................... 3
   1.3 Static HLS ............................ 4
      1.3.1 Basic Synthesis Flow ............... 5
      1.3.2 Challenges ......................... 6
   1.4 Dataflow HLS ......................... 8
      1.4.1 Overview .......................... 9
      1.4.2 Examples ......................... 9
   1.5 Related Work ....................... 12

2 Overview .............................. 16

3 Fluid: An Asynchronous HLS Tool for Complex Program Structures .......... 19
   3.1 Introduction ......................... 19
   3.2 Background ........................... 21
      3.2.1 LLVM ............................ 21
      3.2.2 Dataflow Circuit .................. 22
      3.2.3 Static Token Form .................. 23
   3.3 Fluid Design ......................... 26
      3.3.1 Canonical Form ..................... 27
Acknowledgments

I am deeply grateful to my advisor Prof. Rajit Manohar, who gave me an invaluable opportunity to work with him during my hardest moment. He not only guides me through exciting projects, but also shows me how to do research in a solid and systematic way.

I am also thankful to my other committee members, Prof. Richard Yang and Prof. Abhishek Bhattacharjee, for their insightful suggestions to my research.

I want to thank my collaborators, Lincoln Berkley and Yihang Yang, for their help and contribution to this research.

I want to thank my friends at Yale for their precious friendship, and it is always enjoyable and inspiring to talk to them. I am especially thankful to Ennan Zhai and Mengqi Liu, who helped me get through the challenges during this period.

Lastly, I want to thank my family for their support. And thank you for reading my thesis!
# List of Figures

1.1 The gap in productivity (source: SEMATECH) ........................................... 2  
1.2 HLS overview .................................................................................. 4  
1.3 Example for static HLS .................................................................... 6  
1.4 Pipeline balancing ........................................................................... 7  
1.5 Static scheduling ............................................................................. 8  
1.6 MaxJ example adapted from [14] ....................................................... 10  
1.7 Dataflow computing ......................................................................... 11  
1.8 Dynamic execution of vanilla dataflow HLS ....................................... 12  
2.1 Program partition ............................................................................ 16  
2.2 System overview ............................................................................... 17  
3.1 Fluid synthesis flow: ....................................................................... 20  
3.2 Deterministic dataflow elements ......................................................... 22  
3.3 if example ....................................................................................... 24  
3.4 loop example .................................................................................... 25  
3.5 Canonical form ................................................................................ 27  
3.6 Example for if condition extraction ................................................... 28  
3.7 Example for loop condition extraction .............................................. 29  
3.8 Delivery and collection circuits .......................................................... 33  
3.9 Control token generation .................................................................. 34
3.10 Multi-Path Example ................................................. 35
3.11 Multi-Path Problem and Solution ............................ 36
3.12 Irregular loop and Solution ................................. 37
3.13 Non-deterministic dataflow elements ..................... 40
3.14 Operator clustering example ................................. 42
3.15 SPLIT tree flattening ......................................... 44
3.16 Bundled-data circuit template ............................... 45
3.17 Performance breakdown I .................................... 49
3.18 Performance breakdown II .................................... 51
3.19 Normalized performance for HLS benchmarks: delay (D), area (A), energy (E), leakage power (L), and throughput-inv (I/T) .................................................. 54
3.20 Normalized performance for micro benchmarks: delay (D), area (A), energy (E), leakage power (L), and throughput-inv (I/T) ............................................. 55
4.1 Resource access illustration .................................... 60
4.2 Example code for token network ............................... 64
4.3 Token network template ....................................... 65
4.4 An example program with multiple function calls, and the corresponding extended call graph ..................................... 71
4.5 Token Network example ....................................... 73
4.6 Incremental linking example .................................. 75
4.7 Incremental linker datapath ................................... 77
4.8 Linker Design ...................................................... 78
4.9 Pointer example .................................................. 80
4.10 Pointer argument ................................................. 81
4.11 Incremental compilation example .......................... 82
4.12 Split and merge fusion opportunity ......................... 84
4.13 Normalized per-benchmark performance (I) in default setting: delay ($D$), area ($A$), energy ($E$), leakage power ($L$), and throughput-inv ($1/T$).

4.14 Normalized per-benchmark performance (II) in default setting: delay ($D$), area ($A$), energy ($E$), leakage power ($L$), and throughput-inv ($1/T$).

4.15 Normalized performance (compared to Legup). The dark color represents the default setting for each tool, and the corresponding light color represents the other case.

A.1 Template circuits for token networks.
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Fluid and Fluid-opt Performance</td>
<td>48</td>
</tr>
<tr>
<td>3.2</td>
<td>Performance of LegUp 4.0 and two commercial HLS tools</td>
<td>48</td>
</tr>
<tr>
<td>3.3</td>
<td>Fluid-opt vs CPU implementation</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>PipeLink Analysis</td>
<td>87</td>
</tr>
<tr>
<td>4.2</td>
<td>Normalized Performance (compared to Legup)</td>
<td>88</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Historically, transistor technology scaling provided significant improvements in performance, power, and integration density every few years. This trend has slowed down significantly in today’s sub-10nm regime [1]. To continue satisfying the insatiable needs for high-performance and energy-efficiency, computer system designers have been forced to move from general-purpose microprocessors to FPGAs and both domain-specific and problem-specific accelerators [90, 24, 97, 85, 52, 105, 40, 107, 108, 103, 101, 96].

Designing a system that contains a large collection of accelerators in addition to high-performance CPUs is a non-trivial task. An appealing approach to reducing design complexity is to adopt a design flow using high-level synthesis (HLS), which translates software programs into a hardware description language that implements the same computation. Ideally, HLS can convert the original software implementation that is the target for acceleration into its corresponding hardware implementation [45], and several recent research projects adopt an HLS approach for accelerator design [82, 59, 106, 37]. There are both marketing and technical motivations for using the HLS solutions.

Marketing motivations
1. the productivity gap in ASIC/FPGA design. There is a huge gap between the hardware engineer productivity (i.e., the amount of transistors a hardware engineer can design per month) and the chip complexity. According to Fig. 1.1 from 1981 to 2009, the number of logic transistors per chip has a 58% annual growth rate while the annual growth rate for hardware engineer productivity is only 21%. Such a gap continues to increase as time passes by. As a result, companies are forced to increase the size of their hardware engineering teams.

![Figure 1.1: The gap in productivity (source: SEMATECH)](image)

2. strict requirement on time-to-market (TTM) window. In the consumer electronics market driven by consumer demands for new features and functionalities in the product, a company that launches the product earlier than competitors is likely to gain a bigger share of the market [87]. Therefore, the product team needs to deliver the hardware products in a limited amount of time.

3. the small number of hardware engineers. According to U.S. Bureau of Labor Statistics, in 2020, there were over 1.8 million software engineers [16] while the number for the hardware engineers was only 66K [15].

**Technical motivations**

1. the extensive use of accelerators and heterogeneous SoCs [10]. In mobile devices,
the degree of integration after 2008 keeps increasing to meet the demands of (a) higher computation performance, (b) faster wireless connections, (c) richer multimedia capabilities with heterogeneous components (RF, logic, memory, microelectromechanical systems, etc.). On the datacenter server side, accelerators with new architectures are designed to meet the latency and power requirements. For IoT devices, the conventional Moore’s Law scaling of transistors is not enough to satisfy the extremely low power requirements for computing and communication. To satisfy these requirements, accelerators and SoCs are necessary.

2. dynamic cloud applications. Microsoft team [36] points out that the cloud workloads, which are the popular scenarios to use the accelerators, are rapidly changing in a weekly or monthly manner, which requires the designers to frequently change their ASIC/FPGA designs.

HLS design approach can satisfy these requirements, since it allows the software engineers to generate the hardware accelerators for their programs, and it can effectively increase the hardware engineer productivity and enable them to deliver circuit designs in a shorter time.

1.2 HLS Overview

High-level Synthesis (HLS) is the compilation process which maps high-level algorithms, typically expressed in C and C++, into hardware description language specifying a digital circuit.

As shown in Fig. 1.2, the synthesis is divided into two steps: i) parse the software program and generate the Control-Data Flow Graph (CDFG) which captures the control/data dependencies of the original program. ii) maps the CDFG into either statically scheduled circuit (using statically scheduled HLS) or dataflow circuit (using dataflow HLS).

Need to add: 1) timing info; 2) hardware mapping to the original programs.
CDFG. First of all, the HLS frontend will generate a control flow graph (CFG) [22], which is the standard data structure used for optimizing software programs. Nodes in this graph are basic blocks, which correspond to a collection of consecutive sequential statements with a single entry point and single exit point. Outgoing edges from a basic block correspond to different potential successors, with the successor chosen based on a specified condition. For-loops and while-loops result in cycles in the CFG. Then, for each basic block, a data flow graph (DFG) is generated to capture the data dependencies among instructions within it. A CDFG is the combination of CFG and DFGs for all of the basic blocks.

1.3 Static HLS

The static HLS converts a program into the statically scheduled circuit, which consists of a datapath that contains all the operations (e.g. addition, multiplication, etc.) from the program and a finite state machine that schedules these operations into clock cycles. The state machine serves as a global scheduler that controls the execution sequence of the whole circuit.
1.3.1 Basic Synthesis Flow

The synthesis flow is divided into three stages: scheduling, allocation and binding.

**scheduling.** During the scheduling stage, all operations in the CDFG are statically mapped to clock cycles, which correspond to the states of the state machine. The state machine will activate the states in accordance with the control/data dependencies of the CDFG, which The state machine logically provides the functionality of the program counter, and its scheduling decides the execution sequence of the whole circuit.

There are several scheduling algorithms [75, 104, 65, 88, 63, 55, 56], but they are all NP-hard and require accurate cycle information for each operator before the synthesis. Therefore, it is hard to achieve the “optimal” schedule.

**allocation and binding.** After scheduling, each software operation is mapped to hardware operators, and the intermediate values are assigned to registers (allocation). Since the circuit execution is controlled by the state machine step-by-step, the same hardware operators can be reused by different software operations in different clock cycles. Therefore, the minimum number of operators is determined by the maximum number of corresponding operations scheduled in the same clock cycle, and the minimum number of registers is determined by the maximum number of values sent across clock cycles.

After the allocation, the individual operations and values are mapped to the operators and registers respectively (binding). If an operator (register) is shared among several soft operations (values), multiplexers need to be synthesized to select the correct input values across clock cycles.

**example.** Fig. 1.3(a) shows an example code which has two add operations and one multiply operation, and the variable $y$ is redefined in the two branches of the if statement. Fig. 1.3(b) shows the scheduling result of the static HLS engine. The two if branches (Line 6 and Line 8) are scheduled in $C1$, and the preprocessing (Line 4) and postprocessing (Line 10) codes are scheduled in $C0$ and $C2$ respectively. There are two add operations in different
clock cycles, and the HLS engine allocates one adder and binds the two add operations to it. Fig. 1.3 (c) shows the final circuit, which has one adder and one multiplier. It also has two registers to store the intermediate results from the adder and the multiplier respectively. Furthermore, it has three multiplexers, two of them are used to share the adder in C0 and C1, and the last one is used to select the right y from the if branches.

1. x = ...;
2. c = ...;
3. y = ...;
4. x = x + 1;
5. if (c) {
6. y = x + 2;
7. } else {
8. y = x + 3;
9. }
10. ... y ...

a) example code

1.3.2 Challenges.

The statically scheduled HLS faces several challenges which could largely affect the quality of the generated circuits.

1. The tool requires accurate delay information of each operator before synthesis, which is coupled with specific technology nodes used to implement the circuit, and is not always available.

2. When the source program has if (or nested if) statements, the HLS will trigger the execution of all of the branches regardless of the if conditions. This will increase the energy consumption of the circuit. In Fig. 1.3 both the adder (Line 6) and the multiplier (Line 8) will be triggered when the circuit runs. To make things worse, the circuit has to wait for the slowest branch to finish regardless of the if conditions.

3. The tool schedules all operators into clock cycles (i.e., different pipeline stages), and it is essential that these pipelines have balanced delays. Fig. 1.4 (a) shows an
example DFG which consists of three adders and one multiplier. Suppose the adder delay is 2ns and the multiplier delay is 5ns. There are several ways of scheduling the operators: i) in Fig. 1.4 (b), the stage delay is set to be the adder delay, so the circuit frequency is $500_{MHz}$ and the execution time is 8ns (the multiplier requires 3 stages to finish), ii) in Fig. 1.4 (c), the stage delay is set to be the multiplier delay, so the circuit frequency is $200_{MHz}$ and the execution time is 10ns, iii) in Fig. 1.4 (d), all operators are scheduled in one stage, so the execution time is minimized to 7ns, but the circuit frequency is also the lowest ($143_{MHz}$). Since the clock signal is shared across the whole pipelines, the HLS engine needs to carefully schedule the operators so that each stage does not waste too much time waiting.

4. The tool decides the degree of pipelining before actual execution. Several tools use the directives inserted into the original source program [45, 2, 35, 3, 19] to make better decisions. This relies on the hardware expertise from software programmers. Furthermore, pipelining may be limited due to the lack of runtime information. Consider the example in Fig. 1.5 (a). Inside the for loop, s is conditionally updated (Line 7) based on the value of d (i.e., the values of elements in array A), which is unknown to the HLS engine. As a result, in Fig. 1.5 (b), the scheduler has to reserve time slots for the update to s regardless of the if condition.

5. The whole circuit needs to be re-synthesized on partial changes [60], because the
int A[M];  // assuming the elements in A are: -1, -2, 1, etc.
int i, d;
int s = 0;
for (i = 0; i < 10; i++) {
    d = A[i];
    if (d > 0) {
        /* conditionally executed based on the values of elements in A */
        s += d;
    }
}

static HLS needs to synthesize a state machine that captures the behaviors of all elements in the whole circuit.

1.4 Dataflow HLS

The dataflow HLS converts a program into the dataflow circuit which uses concurrent, parallel dataflow components as hardware building blocks. Unlike the static HLS (Section 1.3), it results in distributed control and an implicit pipelined implementation.

An application is represented as a directed graph, known as a dataflow graph (DFG), of computational blocks with edges defining communication channels between them. The data transferred between blocks is called tokens. When a computation block receives all of the required input tokens from its predecessors, it will operate on the inputs, generate the output and send it to its successor(s). A block remains idle if it does not receive any data.
1.4.1 Overview

The software program organizes the codes in a sequential manner, while the dataflow HLS tries to map it into a concurrent dataflow graph. A lot of tools use domain-specific language (DSL) or directives to specify the algorithm, making it easier to be synthesized. Generally speaking, there are three types of dataflow HLS.

**Domain-specific HLS.** The applications are specified as DFGs in domain-specific dataflow languages which follow the concurrent, message-passing programming models \[86, 4, 14\]. Then, each computation block is synthesized into the RTL description. This approach allows the HLS engine to apply domain-specific optimizations to each computation block individually. Usually these tools target applications with streaming computation and communication patterns (e.g., multimedia applications).

**Directive-based HLS.** Some behavioral HLS tools allow users to represent dataflow computing at the task-level, and use the specific directives to guide the synthesis of the corresponding DFG. The developers are required to identify the dataflow optimization opportunities in the algorithms.

**Vanilla dataflow HLS.** Some HLS tools \[43, 94, 95, 57, 33, 58\] can directly convert the ANSI C/C++ programs into elastic dataflow circuits without any directives or pragmas from the developers. These tools can generate circuits without user involvement.

1.4.2 Examples.

**Domain-specific HLS.** Let’s take MaxCompiler \[68\] as an example. It supports MaxJ, a multiscale dataflow programming language designed for the proprietary dataflow engines in Maxeler Technologies \[13\]. Let’s consider a moving average example which is expressed
Figure 1.6: MaxJ example adapted from [14]

\[
y_i = \begin{cases} 
\frac{x_i + x_{i+1}}{2} & \text{if } i = 0 \\
\frac{x_{i-1} + x_i}{2} & \text{if } i = N - 1 \\
\frac{x_{i-1} + x_i + x_{i+1}}{3} & \text{otherwise}
\end{cases}
\]

Fig. 1.6 shows the implementation of the expressions in MaxJ and the corresponding circuits. It uses Kernel base class to define the kernel functions to be accelerated in the dataflow engines. More specifically, the keywords io, stream, DEFVar, control, constant etc. are reserved to statically map to corresponding circuit components. Users are required to have expertise in multiscale dataflow programming language as well as the architecture details of the proprietary dataflow engines in Maxeler Technologies.


Fig. 1.7 a) shows the example code which uses the DATAFLOW pragma for optimization. The main function calls three kernel functions: getData which fills the contents for
```
void main() {
  #pragma HLS DATAFLOW
  int A[N];
  int B[N];
  getData(A);
  transferData(A,B);
  sendData(B);
}

void getData(int *A) {
  for (int i = 0; i < 10; i++) {
    A[i] = ...;
  }
}

void transferData(int *A, int *B) {
  for (int i = 0; i < 10; i++) {
    B[i] = A[i];
  }
}

void sendData(int *B) {
  for (int i = 0; i < 10; i++) {
    ... use B[i] ...
  }
}
```

Figure 1.7: Dataflow computing
array A, *transferData* which copies the contents from array A to array B, and *sendData* which outputs the contents in array B. Without the *DATAFLOW* pragma, Vivado HLS will execute the three functions sequentially. When the pragma is inserted, however, Vivado HLS would try to pipeline the three functions as much as possible. Fig. 1.7 (b) shows the scheduling without and with the pragma. Note that these three functions have data dependencies on array A and B in the loops, so Vivado will pipeline the sequential loops in the kernel functions if specified.

**vanilla dataflow HLS.** Vanilla dataflow HLS engines [57, 38] can automatically generate pipelined circuits without user involvement. For the example code in Fig. 1.5 (a), it conditionally executes the *if* statement in the loop based on the element values in the array *A*. The dataflow HLS engine could synthesize elastic circuits such that the circuit execution sequence is dynamically decided during runtime based on the array elements, as shown in Fig. 1.8

![Figure 1.8: Dynamic execution of vanilla dataflow HLS](image)

### 1.5 Related Work

There are extensive research efforts [3, 19, 6, 78, 44, 13, 78, 51, 98, 86, 99, 28, 83, 30, 17, 61, 33, 80, 35, 81, 72, 57, 95] for both the static HLS and the dataflow HLS.

Most of the commercial HLS tools are static HLS including C-to-Silicon Compiler [3] from Cadence, Vivado HLS [19] from Xilinx, Intel HLS [9] from Intel, Catapult C [6] from Mentor Graphic and AutoPilot [44] from AutoESL. There are also commercial dataflow HLS including MaxCompiler [13]. Note that MaxCompiler is tightly coupled with the pro-
proprietary platform from Maxeler Technology Ltd, which consists of FPGA-based engines organized in the dataflow manner. These tools typically require users to insert directives and pragmas in the software programs to guide the circuit synthesis [20], and some requires users to explicitly annotate hardware components such as in/out ports in the software programs [5, 13]. They could potentially synthesize efficient circuits, but users are required to have sufficient understanding about circuits as well as the computation/communication patterns of their algorithms. We want to build a HLS tool that is directly usable to software engineers without extra requirements.

A lot of HLS tools are proposed by academia, which falls into two categories: domain-specific tools and the general purpose tools. Usually, the domain-specific HLS tools would require domain-specific languages and apply domain-specific optimizations. For static HLS, there are domain specific tools for the digital signal processing applications such as Impulse-C [78], GAUT [44], Stream-C [51] and ROCCC [98]. The Trident Compiler developed at the Los Alamos National Lab is targeting floating-point scientific applications. Stream-C [51] handles the stream-oriented computing which typically has high-data-rate flow of data sources, fixed-size payloads, compute-intensive operations, access to local memories and synchronization between computational phases. It proposes language constructs, compiler technology and hardware/software libraries to handle it. The GAUT project [44] can not only synthesize the accelerators, but also generate communication and memory units. ROCCC [98] mainly focuses on the parallelization of heavily computation-intensive applications, and it only supports a subset of C language. For example, it only allows properly nested loops accessing integer arrays. Trident compiler [18] focuses on floating-point scientific computations, and it allows users to select the proper floating-point operators from a variety of standard libraries.

For the dataflow HLS, there are HLS tools designed specifically for the stream processing such as CAPH [86], and for multimedia processing such as Open RVC-CAL Compiler (ORCC) [99] and OpenDF [28]. CAPH [86] relies on the actor/dataflow model of computa-
tion, and it describes algorithms as a network of dataflow actors exchanging tokens through unidirectional channels. The behavior of each actor is defined as a set of transition rules using pattern matching. CAL Actor Language [4] is used to specify dataflow computation, and RVC-CAL is a subset of CAL standardized by ISO-MPEG for video coding specifications [29]. ORCC [99] translates applications written in RVC-CAL into an abstract syntax tree and then into IR. Open DataFlow (OpenDF) [28] allows an algorithm to be expressed as a DFG whose edges represent data flow through FIFO channels. The node is written in CAL Actor Language [4]. The HLS engine converts the CAL code in each node into Verilog RTL, which communicates with each other through the FIFO channels.

The general purpose HLS tools are also proposed to handle more general cases. Since the software knowledge does not contain the hardware information (such as concurrency, timing and synchronization constraints, etc.), some works extend the existing languages (usually C language) to let users explicitly add such information in the source code. SpecC [83], Handel-C [30] and SystemC [17] are proposed under the static HLS category, while SPATIAL [61] and Oxigen [80] are proposed for the dataflow HLS. There are also tools that directly work on the vanilla software programs which are usually written in ANSI C/C++. For example, Legup [35], Bambu [81] and Dwarv [72] are proposed as static HLS, while CASH [95], SpatialComputing [33] and DynamicScheduling [57] are proposed as dataflow HLS.

Handle-C allows users to specify clock boundaries in the source code, and SpecC and SystemC enables the explicit specifications of clock edges and events. SPATIAL is based on Delite Hardware Definition Language [62]. It defines hardware accelerators as a hierarchical DFG. Nodes represent the control structures, data operations and memory allocations while edges represent data and effect dependencies. Oxigen takes a compute-intensive C function and translates it to a dataflow representation for the MaxCompiler. The C function is first converted into LLVM IR, which is then optimized with vectorization optimizations. Vectorization allows the user to specify the size of input and output vectors, and changes
the data types of the C function to vector types. By doing that, the MaxCompiler can improve parallelism and fully utilize available bandwidth for each vector element. Legup takes a C program as input, and operates in one of two ways: 1) it synthesizes the entire C program into hardware; 2) it synthesizes the program into a hybrid system comprising a processor (a MIPS soft processor or an ARM core) and one or multiple hardware accelerators. It supports most of the C language features except for dynamic memory allocation and recursive function calls. Dwarv is based on the CoSy compiler framework \cite{8}, so it can take advantage of CoSy’s modular and robust backend. Bambu can generate different Pareto-optimal implementations to do tradeoff between latency and resource requirements. It can also support hardware/software partitioning for designing complex heterogeneous platforms. CASH and SpatialComputing can map C program into asynchronous dataflow circuit, which is a good fit for the dataflow computation model. DynamicScheduling extends the static HLS methodology and enables dynamic scheduling of circuits.
Chapter 2

Overview

In this work, we will build an asynchronous dataflow HLS tool that synthesizes fully-pipelined circuits for the general programs. We consider the asynchronous circuit as a natural choice for implementing the dataflow circuit. Furthermore, dataflow circuits can be automatically pipelined because they get rid of the global control; ironically, they are hard to be fully pipelined due to the lack of such global control, and this work will fill the gap. Note that certain complex program patterns are hard to be synthesized into fully pipelined dataflow circuits, and this work can support these patterns which are common in real-world applications.

Generally speaking, a full program consists of multiple functions which could call each other.

```
1 void m() {
2   ...
3   y = f(x);
4   ...
5 }
6
7 int f(...)
8   ...
9 }

1 void m() {
2   ...
3   y = f(x);
4   ...
5 }
6
7 int f(...)
8   ...
9 }
```

a) full program     

b) part 1: w/o function call     

c) part 2: function call

Figure 2.1: Program partition
Consider the code in Fig. 2.1(a) which has two functions \( m \) and \( f \). \( m \) calls \( f \) in the function body, and \( f \) is a single-input-single-output function. We could divide the program into two parts: in part 1, it has all of the code except the ones involving function calls (Fig. 2.1(b)); in part 2, it only has the function call codes (Fig. 2.1(c)). We propose Fluid (Chapter 3) to synthesize the codes in part 1, and PipeLink (Chapter 4) to synthesize the codes in part 2.

![Fluid Illustration](image-a)

![PipeLink Illustration](image-b)

Figure 2.2: System overview

Fig. 2.2 gives an overview of our system. Fluid (Fig. 2.2(a)) maps each function into a pipelined circuit. Since there is no function call involved, different functions are mapped into independent circuits. Furthermore, Fluid would synthesize dangling ports for each function’s I/O ports if there is any as well as for the arguments (e.g., \( x \)) and return values (e.g., \( y \)) involved in the function call. Then, PipeLink (Fig. 2.2(b)) will take over and handle the function calls. It essentially connects together the dangling ports that are generated by Fluid. Our system would partition any program into two parts based on the function calls, and synthesize the circuits for each part. Circuit synthesis could be divided into compilation stage and linking stage accordingly.

This thesis has made the following contributions:

- It proposes a systematic way to synthesize the fully pipelined asynchronous dataflow circuits. The circuit synthesis can be divided into two stages: for the compilation stage, it proposes Fluid (Chapter 3), which could handle complex program structures that are not supported by the existing tools; for the linking stage, it proposes PipeLink (Chapter 4), which supports pipelined function calls that are previously unavailable.
- It supports more features including pipelined memory access, function pointers (Section 4.5.1) and separate compilation (Section 4.5.3).

- It proposes a bunch of dataflow optimizations (Section 3.5 and Section 4.6) that could effectively improve the dataflow circuit quality.

Furthermore, we implement our system on LLVM [11], which is a widely-used open-source compiler framework whose technologies are also used in Apple’s production compiler. Therefore, we could leverage the existing optimization efforts from the software compiler community. We compared our system with a widely-used academic tool and two commercial tools, so the performance of our system is well-studied. Lincoln Berkley built a standalone dataflow graph optimizer in C++ and Yihang Yang designed the bundled-data template circuits for this work.
Chapter 3

Fluid: An Asynchronous HLS Tool for Complex Program Structures

In this chapter, we present an HLS methodology tailored to high-performance asynchronous dataflow circuits building on prior work in dataflow synthesis [93]. Furthermore, we propose a new solution to dataflow circuit generation needed when translating real-world programs with complex control flows. We implement our approach in the LLVM compiler framework [11], and show that our generated circuits achieve better performance in throughput and energy compared to a number of existing HLS tools. We also quantify the benefits of dataflow graph optimizations on the quality of the generated circuits.

3.1 Introduction

There has been significant activity in translating behavioral descriptions of asynchronous computations into asynchronous circuits, and the majority of these efforts focus on translating a concurrent, message-passing programming language into asynchronous circuits [73, 53, 42, 54, 100, 92, 102]. There has also been previous work in translating software programs into asynchronous circuits [33, 95]. Furthermore, some synchronous HLS tools also synthesize latency-insensitive dataflow circuits [43, 94, 57, 58]. Other tools use domain-specific languages and special pragmas to simplify the high-level synthesis prob-
The most complex aspect of generating dataflow circuits is managing conditional execution and conditional generation of tokens. Prior work either mostly avoids conditional tokens, or only supports conditional tokens for simple control structures.

This chapter presents Fluid, a HLS tool that translates C programs into asynchronous dataflow circuits. Our work extends existing dataflow synthesis techniques to a wider class of software programs by supporting complex control-flow structures that naturally occur in software. This also permits us to use optimizations that might create complex control structures. We also incorporate optimizations that operate directly on the dataflow graph structure, further improving our results.

Fig. 3.1 shows the overall flow of our tool. Fluid goes through a number of steps to translate C programs to asynchronous dataflow circuits. Starting from the C program, we use the LLVM compiler framework [11] to generate the LLVM IR (Intermediate Representation). Then, LLVM would apply the optimization passes and produce the optimized IR, which is a CDFG (Control-Data Flow Graph) that captures the control and data dependencies of the original program. Fluid analyzes the IR and rewrites the CFG into canonical form to handle complex control structures. After this, Fluid synthesizes the dataflow circuit
which is represented as a dataflow graph. Then, it applies several dataflow optimizations to produce an optimized dataflow graph. Then, the optimized dataflow graph is mapped to asynchronous bundled-data circuits through technology mapping. Lastly, we built an asynchronous circuit simulator to simulate the generated circuits and get the computation results and performance numbers. We compare Fluid against an academic HLS tool (LegUp [35]) and two different commercial HLS tools on a combination of micro-benchmarks and public HLS benchmarks. Our results show that Fluid can increase the circuit speed by 2X, reduce the energy consumption by 5X and increase the throughput by 2.5X.

Our contributions are: (i) an asynchronous HLS tool that translates C to an asynchronous dataflow circuit with results that are significantly superior to an academic HLS tool and outperform commercial HLS tools on throughput and energy; (ii) a new technique for dataflow graph construction in the presence of complex control flow; and (iii) a collection of dataflow graph optimizations that improve the quality of the final implementation. The remaining chapter is organized as follows: Section 3.2 introduces the prior work that we build on. Section 3.3 presents how Fluid constructs dataflow graphs based on the control-flow graph (CFG), including support for irregular CFGs (Section 3.3.5). Section 3.5 describes the dataflow optimizations currently used in Fluid. Section 3.6 evaluates Fluid against three other HLS tools. We provide an overview of the large body of related work in Section 3.7.

3.2 Background

3.2.1 LLVM

To translate C programs, we leverage the production-quality LLVM open-source compiler framework [11]. The LLVM front-end translates different programming languages into a common intermediate representation (IR). LLVM also includes a large number of optimization passes that re-write and improve the quality of the IR from a software perspective [11].
The standard data structure used for optimizing software programs is the control flow graph (CFG) [22]. Nodes in this graph are basic blocks, which correspond to a collection of consecutive sequential statements with a single entry point and single exit point. Outgoing edges from a basic block correspond to different potential successors, with the successor chosen based on a specified condition. For-loops and while-loops result in cycles in the CFG.

### 3.2.2 Dataflow Circuit

Dataflow circuit consists of dataflow elements that communicate with their predecessors and successors through handshake protocols, thus the circuits do not require global control. Dataflow elements remain idle until they receive tokens [47]. The circuits can be implemented with either synchronous [77, 43, 94, 57, 38, 86, 28, 4, 84, 39, 21, 48, 67, 80] or asynchronous logic [95, 33, 93].

![Deterministic dataflow elements](image)

In our work, we build our dataflow circuits using the eight basic dataflow elements shown in Fig. 3.2:

- **COPY**. It receives one input token and duplicates it to multiple output ports. More specifically, it will not receive new input tokens until all output tokens have been acknowledged by the successors.
• **FUNC.** It receives input tokens, computes some function (e.g., add, multiply, divide, shift, modulo, compare) of the received values, and sends the results to the output port(s).

• **MERGE.** It receives multiple inputs from input ports and one control token $c$ from the control port, and then sends one of the inputs to the output port based on the control token value. In this work, we assume that the left (right) port is selected under the false (true) condition.

• **SPLIT.** It receives one input from the input port and one control token from the control port, and then sends it to one of the output ports based on the control token value. Similar to **MERGE**, we assume that the left (right) port is selected under the false (true) condition.

• **SINK.** It is used to absorb the unused tokens in the circuit.

• **SOURCE.** It generates tokens with pre-configured constant values and sends it to the output port.

• **BUF.** It is a one-place FIFO.

• **INIT.** It is a **BUF** initialized with a data token (configured during reset).

Any dataflow graph that is constructed using these elements is guaranteed to be deterministic, and has the property of slack elasticity. Slack elasticity provides a theoretical guarantee of correct behavior under a wide range of pipelining options [69].

### 3.2.3 Static Token Form

Our work builds on previous efforts to translate hardware description languages to dataflow asynchronous circuits. In particular, the static token form (STF) representation was introduced to translate the CHP hardware description language into a dataflow graph [93]. The
CHP language was translated into a CDFG, and variables with multiple definitions (for example, the left hand side of an assignment statement) and uses (for example, the right hand side of an assignment statement) in CHP were re-written into the canonical STF form. Informally, STF guarantees that the conditions that cause a variable to be defined match the condition under which it is used; this permits variables to be replaced by channels, and values become tokens in the dataflow graph [93]. STF combines concepts from both static single assignment (SSA) [46] form and static single information (SSI) [23] form into a unified analysis. If a variable is conditionally used, STF inserts a split instruction to conditionally create a new token for the conditional use; if a variable comes from multiple definitions under various conditions, STF inserts a merge instruction to conditionally select from multiple defs and assign the new token to the variable.

It is fairly straightforward to convert sequential statements into static token form. The key is to handle the if case and the loop case.

**if case.** Fig. 3.3 (a) shows the same if statement as the one in Fig. 1.3 (a). $x$ is redefined at Line 4 and a new version of $x$ is generated. Similarly, $y$ is also redefined at Line 6 and Line 8 respectively. To satisfy the SSA requirement, STF in Fig. 3.3 (b) labels $x$ as $x_1$ (at Line 4) and $y$ as $y_1$ (at Line 8) and $y_2$ (at Line 10). Furthermore, $x_1$ is conditionally used in both branches, so the STF contains two split instructions to conditionally generate

```plaintext
1 x = ...;
2 c = ...;
3 y = ...;
4 x = x + 1;
5 if (c) {
6     y = x + 2;
7 } else {
8     y = x + 3;
9 }
10 ... use y ...
```

**Figure 3.3: if example**

at Line 4 and a new version of $x$ is generated. Similarly, $y$ is also redefined at Line 6 and Line 8 respectively. To satisfy the SSA requirement, STF in Fig. 3.3 (b) labels $x$ as $x_1$ (at Line 4) and $y$ as $y_1$ (at Line 8) and $y_2$ (at Line 10). Furthermore, $x_1$ is conditionally used in both branches, so the STF contains two split instructions to conditionally generate
and $x_3$ for the uses at Line 8 and Line 10 respectively. Lastly, after the if statement, $y$ comes from either the true branch ($y_1$) or the false branch ($y_2$), so STF inserts a merge instruction to conditionally select $y_1$ and $y_2$ based on the condition $c$ and assign the value to $y_3$ (not $y$). Fig. 3.3(c) shows the synthesized circuit for the if example. The split operator conditionally generates token $x_2$ and $x_3$ from $x_1$, and the merge operator conditionally selects from $y_1$ and $y_2$ and assigns the new token to $y_3$.

**loop case.** Fig. 3.4(a) shows the loop statement, which runs the loop until the loop counter $x$ is greater or equal than 10. The loop condition ”$x < 10$” (Line 2) uses variable $x$ that comes from two definitions: initial $x$ (Line 1) before entering the loop, and updated $x$ (Line 4) during the execution of the loop. Therefore, in Fig. 3.4(b), the STF contains a merge instruction (Line 3) to conditionally select $x$ in the loop header block. Then in the loop body block, a new loop counter $x_2$ is generated. In the loop exit block, loop condition $c$ is updated based on the value of $x_2$, and a new version of counter $x_3$ is conditionally generated if the loop continues (i.e., $c$ is true).

The challenge comes from the loop condition $c$ that is used in the merge instruction (Line 3), because it is used (Line 3) before being generated (Line 7). During the initial execution of the loop, the merge instruction requires a control token that is generated from its output, and this is a dependency deadlock. To break the deadlock, the STF attaches an init instruction (with initial value 0) coupled with the merge instruction, so the merge instruction has an initial control token 0 before the loop starts. As a result, the merge
instruction would select the initial $x$ before entering the loop. The final circuit is shown in Fig. 3.4 (c). When the loop finishes, the loop condition "$x < 10" becomes 0, which is stored in the \textit{INIT} operator. In this case, the circuit restores to the initial state and is ready for the next run.

STF does no optimizations to the synthesized dataflow circuits, and it only deals with CHP programs with simple control structures.

3.3 Fluid Design

Fluid is written as a compiler pass within the LLVM framework. It takes an optimized IR as input, and re-constructs the CFG for the program. While LLVM optimization passes improve the quality of the IR, they can re-structure the CFG. To convert this optimized CFG to static token form, we introduce new techniques discussed below that modify the CFG and IR so that the resulting program is equivalent to the original IR, and can be readily converted into static token form. Finally, we apply dataflow circuit optimizations to obtain the final circuit.

IR constructs that perform computation (e.g. addition, division, etc.) can be translated into dataflow function blocks in the usual manner \cite{93}. The challenging part of STF generation is creating the \textit{SPLIT} and \textit{MERGE} circuits correctly, along with their control flow conditions. We focus on this aspect below.

As discussed above, STF requires that a variable definition (a "def") and use occur under the same condition. After the CFG is constructed using standard techniques \cite{22}, Fluid computes the def-condition and use-condition for each variable in the program. If the def-condition and the use-condition for a variable are different, Fluid constructs a \textit{delivery circuit} to create a conditional copy of the variable; symmetrically, it constructs a \textit{collection circuit} that conditionally selects the correct version of the variable from multiple conditional definitions of the variable. We detail this process below.
Our analysis makes extensive use of the standard compiler notion of dominators. A basic block $A$ dominates $B$ if every control flow path from entry to $B$ must pass through $A$. A basic block $B$ post-dominates $A$ if every path from $A$ to the exit must pass through $B$. The immediate dominator for a basic block is its closest dominator (apart from itself) in the control flow graph.

### 3.3.1 Canonical Form

To simplify the design of Fluid, we impose a canonical form requirement on CFGs which consists of two parts:

- every loop has a single-entry and single-exit point;
- every if-block is properly-nested;

The first constraint means that we could treat each loop as one super block which is single-in-single-out as shown in Fig. 3.5 (a). The second constraint means that we could treat each if as a single-in-single-out super block as shown in Fig. 3.5 (b). Note that each block (loop_header, loop_body, loop_exit, if_header, if_exit) itself could be another if or loop block, which also satisfies the canonical form.

Section 3.3.5 provides techniques to handle a commonly occurring class of non-canonical CFGs. The loop constraint means we can handle all loop-carried dependencies (back edges)
using the technique in [93], and ignore those edges in the CFG for condition extraction below.

### 3.3.2 Condition Extraction

Fluid is built upon Static Token Form, which needs to compute the def/use conditions of each variable. These conditions are the same as the entering conditions of the basic block where the variable lives in. Therefore, given a CFG, Fluid first extracts the conditions into each basic block from other basic blocks. Since the canonical CFG has single-entry/single-exit loops, we can safely divide the entire CFG into smaller regions: inside each small region (corresponding to a if-block or a loop-block), it has one enter (exit) block that dominates (post-dominates) all the internal blocks.

Given a block, Fluid uses reverse breadth-first-search along its predecessors to explore all the paths into this block. The search stops when it encounters the immediate dominator of this block. Furthermore, if the search encounters a loop, it will treat the whole loop block as one super block and directly skip it (i.e., without going inside the loop block).

**if condition extraction.** Fig. 3.6 (a) shows a CFG for a *if* example. This CFG can be divided into two smaller regions: \( \{B_1,B_2,B_3,B_4,B_5\} \) and \( \{B_0,B_1,B_2,B_3,B_4,B_5,B_6\} \). A block can
belong to multiple regions, and Fluid assigns it to the smallest region. Note that \( \{B_3,B_4,B_5\} \) is not considered as an independent region, because \( B_2 \) does not dominate \( B_5 \). The CFG has three branching variables \( c_0, c_1 \) and \( c_2 \) in \( B_0, B_1 \) and \( B_2 \) respectively.

Consider \( B_5 \) as the target block. In the first step (Fig. 3.6 (b)), Fluid explores the direct predecessors of \( B_5 \): \( \{B_1,B_3,B_4\} \). The conditions for \( B_3 \rightarrow B_5 \) and \( B_4 \rightarrow B_5 \) are both empty, and \( B_1 \rightarrow B_5 \) is \( \{c_1=1\} \). In the second step (Fig. 3.6 (c)), Fluid further explores \( \{B_3,B_4\} \)’s predecessor \( B_2 \). Note that \( B_1 \) is the immediate dominator of \( B_5 \), so the search stops at \( B_1 \).

Now Fluid records the conditions for \( B_2 \rightarrow B_5 \): \( \{c_2=0\}, \{c_2=1\} \). In the third step (Fig. 3.6 (d)), Fluid explores \( B_2 \)'s predecessor \( B_1 \), and updates the conditions for \( B_1 \rightarrow B_5 \) to be: \( \{c_1=1\}, \{c_1=0,c_2=0\} \) and \( \{c_1=0,c_2=1\} \).

**loop condition extraction.** Fig. 3.7 (a) shows a CFG for a loop example. This CFG consists of two smaller regions: \( \{B_1,B_2,B_3,B_4\} \) as a loop super block, and \( \{B_0,B_1,B_2,B_3,B_4,B_5\} \). The loop entering condition is \( c_1 = 1 \), and the loop super block has a nested if-block controlled by \( c_2 \). The loop exit condition is \( c_3 = 0 \).

Consider \( B_5 \) as the target block. In the first step (Fig. 3.7 (b)), Fluid explores the direct predecessors \( B_0 \) and \( B_4 \), and update the conditions for \( B_0 \rightarrow B_5 \) and \( B_4 \rightarrow B_5 \) respectively. In the

---

\[1\] For all of the CFGs in this thesis, we assume the left branch is the false branch, and the right branch is the true branch.

---

Figure 3.7: Example for loop condition extraction.
second step (Fig. 3.7(c)), Fluid tries to further explore \( \{B_0, B_4\} \)'s predecessors. Since \( B_0 \) is the immediate dominator of \( B_5 \), the search stops at \( B_0 \). As for \( B_4 \), it belongs to a loop super block which is abstracted as single-in-single-out super block, so Fluid would consider \( B_0 \) (not \( B_2 \) or \( B_3 \)) as \( B_4 \)'s predecessor, and the condition for \( B_0 \rightarrow B_4 \rightarrow B_5 \) is \( \{c_1=1\} \). Note that if the loop super block is in the middle of the search path, then the loop condition \( (c_3) \) does not appear in the condition chain \(^2\) however, if the loop exit block \( B_4 \) is the source node of the search path (e.g., \( B_4 \rightarrow B_5 \)), then the loop condition should be kept in the condition chain.

The merging operation. A CFG can have multiple paths between two basic blocks, each corresponding to a chain of conditions. However, two condition chains can be merged if they only have one different condition, and the two different conditions are complementary\(^3\). Fig. 3.6 has two condition chains from \( B_2 \) to \( B_5 \): \( \{c_2=0\} \) along \( B_2 \rightarrow B_3 \rightarrow B_5 \), and \( \{c_2=1\} \) along \( B_2 \rightarrow B_4 \rightarrow B_5 \). \( c_2 \) is complementary in the two chains, so the merged chain has condition \( \{} \), and we can collapse the two paths treating it as a single virtual path \( B_2 \rightarrow B_5 \). We repeatedly apply this merge operation, until no paths can be merged.

**Theorem 1.** In a canonical CFG, if there are multiple merged paths for \( src \rightarrow dst \), then \( dst \) cannot post-dominate \( src \).

**Proof.** For loop-free segments of the CFG, we prove the result by contradiction. Suppose \( dst \) post-dominates \( src \). Any path from \( src \) that adds conditions of the form \( c_i = 0 \) or \( c_i = 1 \) must also have a branch that includes the other condition, and they must all re-converge prior to/at \( dst \) since \( dst \) post-dominates \( src \). Hence, all possible conditions associated with paths from \( src \) to \( dst \) exist, and they can be merged into one path \( src \rightarrow dst \). This contradicts our multi-path assumption, so \( dst \) cannot post-dominate \( src \).

If \( src \) and \( dst \) are within the same loop, we can repeat the argument above for the

---

\(^2\)We assume that the source program does not have a dead loop. If there is dead loop, then Fluid can be extended to handle this case by always including the loop condition in the condition chain when the loop super block appears in the search path

\(^3\)In if-statements and loops, we have two-way branching so this is naturally satisfied. To simplify our presentation, we assume multi-way branches like those from switch statements are transformed into two-way branches. The extension to multi-way branches is straightforward.
sub-CFG that only includes the loop body. Otherwise suppose \( src \) belongs to \( Loop_1 \) and \( dst \) belongs to \( Loop_2 \). Since all loops are single-entry-single-exit, we divide \( src \rightarrow dst \) into \( src \rightarrow Loop_1exit \rightarrow Loop_2entry \rightarrow dst \), and repeat the argument for each segment. The case when only one of \( src/dst \) belonging to a loop is similar.

**Lemma 2.** If \( dst \) unconditionally connects to \( dst_2 \), then the number of merged paths for \( src \rightarrow dst \) equals that for \( src \rightarrow dst_2 \).

**Proof.** Since \( dst \) connects to \( dst_2 \) unconditionally, \( dst \) is the immediate dominator of \( dst_2 \). We can divide \( src \rightarrow dst_2 \) into \( src \rightarrow dst \) and \( dst \rightarrow dst_2 \). The condition for \( dst \rightarrow dst_2 \) is \( \{\} \), so the conditions and the number of merged paths for \( src \rightarrow dst_2 \) match that for \( src \rightarrow dst \).

Our main result that is the basis for generating static token form is the following:

**Theorem 3.** Given two basic blocks \( src \) and \( dst \) in a canonical CFG, there is at most one merged path from \( src \) to \( dst \).

**Proof.** If \( dst \) is not reachable from \( src \), then there is no path between them and we are done. If \( dst \) has one predecessor, then we traverse the CFG backward until we reach a basic block with multiple predecessors, or we reach \( src \). If we reach \( src \), the result trivially holds. Otherwise, call the new basic block \( dst' \). By Lemma 2, the merged path count from \( src \) to \( dst' \) matches \( src \) to \( dst \).

\( dst' \) has more than one direct predecessor and is reachable from \( src \). Suppose its immediate dominator is \( iDom \). By the canonical form assumption, \( dst' \) post-dominates \( iDom \). Also, any path from the program entry that contains \( src \) and \( dst' \) must include \( iDom \). If that path has \( iDom \) before \( src \), then \( iDom \) must also dominate \( src \); otherwise we would have found a path from program entry to \( src \) to \( dst' \) without \( iDom \)—a contradiction. Hence, there are two cases:

**Case 1:** \( iDom \) dominates \( src \), then \( dst' \) post-dominates \( src \) as well. According to Theorem 1, there exists only one merged path for \( (src,dst') \), and the proof is done.
Case 2: $iDom$ does not dominate $src$, in which case $iDom$ must be on any path from $src$ to $dst'$; we divide $src \to dst'$ into two parts: $src \to iDom$ and $iDom \to dst'$. There is one merged path for $iDom \to dst'$, so we truncate $src \to dst'$ to $src \to iDom$. By repeating this, we eventually reduce Case 2 to Case 1.

3.3.3 Delivery and Collection Circuit Construction

If a token is defined in $src$ block and used in $dst$ block, Fluid will construct the delivery circuit to conditionally propagate this token.

Calculating the delivery conditions. In Section 3.3.2 Fluid records $nestedConds$, which contain the entering conditions into $dst$ from its predecessors that are dominated by its immediate dominator $iDom$. Next, Fluid needs to calculate the delivery conditions for $src \to dst$. The algorithm is shown in Algorithm 1.

```
input : src, dst, nestedConds
output: deliveryConds
if nestedConds.find(src, dst) then
    return nestedConds.get(src, dst);
end
extract dst block’s immediate dominator, $iDom$;
conditions = nestedConds.get(iDom, dst);
return getDeliveryConds(src, iDom, nestedConds).append(conditions);
```

Algorithm 1: getDeliveryConds(src, dst, nestedConds)

If $src \to dst$ can be found in $nestedConds$, Fluid can directly get the delivery conditions from it. Otherwise, Fluid calculate the immediate dominator $iDom$ of $dst$, divide $src \to dst$ into $src \to iDom$ and $iDom \to dst$. The conditions for $iDom \to dst$ is known in $nestedConds$, so we just need to calculate the conditions for $src \to iDom$ by applying the same rule iteratively. Based on Theorem there exists only one condition chain for any $src \to dst$, so we can simply append these conditions together to form the final conditions.

Synthesizing the delivery circuit. Fluid synthesizes $SPLITS$ for each unique condition variable in the delivery conditions and connects them following the same order.
Fig. 3.8 shows the same CFG as in Fig. 3.6 (a). $x$ is defined in $B_0$ and used in $B_2$, and the delivery condition for $B_0 \rightarrow B_2$ is $\{c_0=0, c_1=0\}$. In Fig. 3.8(b), Fluid synthesizes $SPLIT_0$ (in $B_0$) to generate $x_1$ for $B_1$ when $\{c_0=0\}$, and $SPLIT_1$ (in $B_1$) to generate $x_2$ for $B_2$ when $\{c_1=0\}$.

**Synthesizing the collection circuit.** If token $y$ in $dst$ has multiple reaching definitions $y_1, y_2, ..., y_n$ in $dst$’s $n$ predecessors, Fluid synthesizes the collection circuit to pick the right token. Suppose $dst$’s immediate dominator is $iDom$. Starting from $iDom$, the program will traverse through different paths into $dst$’s predecessors before entering into $dst$. The conditions associated with each traversal are the collection conditions for the corresponding predecessor. Then, Fluid synthesizes $MERGES$ for each unique condition variable in the collection conditions and connects them in the reverse order of the collection conditions.

In Fig. 3.8(a), $B_5$ receives $\{y_0, y_1, y_2\}$ from $\{B_3, B_4, B_1\}$, and assigns the final value to $y$. The collection conditions are:

$$B3 \rightarrow B_5: \{c_1=0, c_2=0\} \quad B4 \rightarrow B_5: \{c_1=0, c_2=1\} \quad B1 \rightarrow B_5: \{c_1=1\}$$

Fig. 3.8(c) shows the synthesized $MERGE$ tree in $B_5$. 
### 3.3.4 Control Token Generation

The delivery and collection circuits consist of *MERGES* and *SPLITS* that require the control tokens, which could also need delivery/collection circuits if used/defined conditionally.

In Fig. 3.8a, suppose $c_0$, $c_1$ and $c_2$ are all defined in $B_0$. Then Fluid will conditionally generate $c'_1$ for $B_0 \rightarrow B_1$ (Fig. 3.9a) and $c'_2$ for $B_0 \rightarrow B_2$ (Fig. 3.9b) as well as the new delivery circuit for $B_0 \rightarrow B_2$ (Fig. 3.9c) and the collection circuit for $B_5$ (Fig. 3.9d).

### 3.3.5 Handling Non-canonical CFGs

**Multi-Path problem.** We handle the case where there are multiple merged paths for $src \rightarrow dst$.

Fig. 3.10a shows a CFG with four basic blocks, and $c_0$ and $c_1$ are the condition variables for $B_0$ and $B_1$ respectively. $x_0$ is defined in $B_0$ and used in $B_2$, so it needs a delivery circuit.

Fig. 3.10b shows the delivery circuit for $B_0 \rightarrow B_2$. We would create *SPLIT*$_0$ (in $B_0$) to conditionally generate $x_1$ (for $B_1$) and $x_2$ (for $B_2$). We also need *SPLIT*$_1$ (in $B_1$) to conditionally generate $x_3$ (for $B_2$). $B_2$ has two incoming tokens: $x_2$ with collection condition $\{c_0=1\}$, and $x_3$ with collection condition $\{c_0=0, c_1=1\}$. Hence a *MERGE*$_0$ and *MERGE*$_1$ are needed to select them. However, $c_0$ and $c_1$ are defined in $B_0$ and $B_1$ respectively, so *MERGE*$_0$ and *MERGE*$_1$ (in
In Fig. 3.10c, we attempt to introduce \( \text{split}_2 \) to conditionally propagate \( c_1 \) to \( B_2 \). However, \( c_0 \) is defined in \( B_0 \) and used in \( B_2 \), and it requires the delivery circuit for \( B_0 \rightarrow B_2 \)—the same circuit we were attempting to construct for \( x \)! Hence, the standard approach to constructing a dataflow graph fails if there are multiple paths after the merging operation.

![CFG](a) CFG.

![Strawman delivery circuit](b) Strawman delivery circuit.

![Dilemma in building the delivery circuit](c) Dilemma in building the delivery circuit.

Figure 3.10: Multi-Path Example.

Fig. 3.11a illustrates the multi-path problem in a CFG. A broken line means there exist paths between two blocks, and the solid line is a direct connection. Assume there are multiple paths for \( \text{src} \rightarrow \text{dst} \) which cannot be merged into one path. By Theorem 3.1, \( \text{dst} \) cannot post-dominate \( \text{src} \), so it cannot post-dominate its \( n \) direct predecessors between \( \text{src} \rightarrow \text{dst} \)
Figure 3.11: Multi-Path Problem and Solution.

either. We partition these $n$ predecessors into two sets: \{pred$_1$, ..., pred$_m$\} which are post-dominated by dst, and \{pred$_{m+1}$, ..., pred$_n$\} which are not post-dominated by dst. Let iPdom be the immediate post-dominator of src. Then there exists paths between \{pred$_{m+1}$, ..., pred$_n$\} and iPdom without passing through dst. Among these paths, suppose the direct predecessors of iPdom are \{PRED$_{m+1}$, ..., PRED$_n$\}.

Fig. 3.11b is our proposed solution. The main idea is to modify the CFG and introduce a new basic block that post-dominates src. This new block fakeBB replaces dst, i.e., all of dst’s predecessors \{pred$_1$, ..., pred$_n$\} now point to fakeBB directly. In addition, we also modify \{PRED$_{m+1}$, ..., PRED$_n$\} to point to fakeBB directly. Thus, fakeBB now post-dominates src, and there will be only one merged path for src $\rightarrow$ fakeBB.

To preserve the correctness, we add edges from fakeBB to iPdom and dst respectively, and a fresh condition variable flag which takes inputs from its direct predecessors \{pred$_1$, ..., pred$_n$.PRED$_{m+1}$, ..., PRED$_n$\}.

If flag is true, fakeBB jumps to dst; if flag is false, fakeBB jumps to iPdom.

\{pred$_1$, ..., pred$_n$\} have direct connections to dst, so these blocks will propagate token 1 to flag following the same conditions, making fakeBB jump to dst. Similarly, if \{pred$_{m+1}$, ..., pred$_n$\} jumps to \{PRED$_{m+1}$, ..., PRED$_n$\}, they will propagate 0 to the flag, making fakeBB jump to

Note that dst could have other predecessors that do not belong to the paths for src $\rightarrow$ dst, and they are not relevant to the multi-path problem.
The modified CFG has the same behavior as the original one.

**Theorem 4.** The above transformation reduces the number of \((src, dst)\) pairs that cause the multi-path problem in a CFG.

**Proof.** As Fig. 3.11b shows, we added a new block \(fakeBB\), and three groups of new connections: \(\{PRED_{m+1}, \ldots, PRED_n\} \rightarrow fakeBB, fakeBB \rightarrow iPdom\) and \(fakeBB \rightarrow dst\). Since \(fakeBB\) post-dominates \(src\), there exists only one merged path for \(src \rightarrow fakeBB\). Furthermore, \(fakeBB\) directly connects to \(iPdom\) and \(dst\), which does not change the post-dominance relationship between \(dst\) and \(iPdom\), so \(fakeBB\) does not introduce a new multi-path pair. Therefore, our solution can eliminate one multi-path pair \((src \rightarrow dst\) in Fig. 3.11a). □

**Irregular Loops.** The canonical loop block Fig. 3.5(a) requires that each loop block has one loop header block and one loop exit block. Now we handle the case where a loop has more than one exit block and (or) more than one header block.

Fig. 3.12a shows the irregular loop with more than one exit block. When the loop
condition variable \textit{loopVal} is 1, the loop exit block \textit{Loop} exits the loop and jumps to its successor set \{\textit{suc}_1\}; otherwise the loop continues. However, starting from \textit{BB}_{\text{diverge}}, there is a second exit block \textit{Loop}_2.

Fig. 3.12b shows the solution. We create two new blocks \textit{fake} and \textit{fake}_{\text{suc}}. The new loop condition variable is \textit{flag}. If \textit{Loop}_2 is executed, \textit{flag} becomes 1 and the loop exits; otherwise \textit{flag} equals to \textit{loopVal}. Therefore, the new \textit{CFG} has the same behavior of running/exiting the loop as the original.

When the loop exits, \textit{fake} jumps to \textit{fake}_{\text{suc}}. If the loop exits from \textit{Loop}_1, \textit{flag} equals to 0 and \{\textit{suc}_1\} will be executed. If the loop exits from \textit{Loop}_2, \textit{flag} equals to 1 and \{\textit{suc}_2\} will be executed. Therefore, the new \textit{CFG} has the same behavior after exiting the loop as the original one.

Fig. 3.12c shows the irregular loop with more than one header block. The regular loop path is \textit{Loop}_{\text{pred}_1} \rightarrow \textit{Loop}_{\text{header}_1} \rightarrow \textit{BB}_{\text{merge}} \rightarrow \textit{Loop}_1 \rightarrow \textit{Loop}_{\text{header}_1}. However, there is a second path that enters the loop: \textit{Loop}_{\text{pred}_2} \rightarrow \textit{Loop}_{\text{header}_2} \rightarrow \textit{BB}_{\text{merge}}, which results in a non-canonical loop block.

Fig. 3.12d shows the solution. We create a fake block \textit{Loop}_{\text{fake header}} as the actual loop header block, and the previous loop header predecessors, \textit{Loop}_{\text{pred}_1} and \textit{Loop}_{\text{pred}_2}, both connect to the fake block. Starting from this fake block, the CFG goes to the previous loop headers \textit{Loop}_{\text{header}_1} and \textit{Loop}_{\text{header}_2} depending on a new condition variable \textit{flag}. If \textit{flag} becomes true, the CFG jumps to \textit{Loop}_{\text{header}_1}, otherwise it jumps to \textit{Loop}_{\text{header}_2}. The remaining blocks are not changed. By doing this, the new loop block has only one ”fake” loop header \textit{Loop}_{\text{fake header}}. Note that \textit{Loop}_{\text{header}_2} could connect to other successors \{\textit{suc}_2\} other than \textit{BB}_{\text{merge}}, which results in a multi-exit loop. This can be solved using the previous algorithm.

The \textit{flag} generation is non-trivial though. It takes values from \textit{Loop}_{\text{pred}_1} and \textit{Loop}_{\text{pred}_2}: when \textit{Loop}_{\text{pred}_1} is executed, it propagates token 0 to the \textit{flag}; when \textit{Loop}_{\text{pred}_2} is executed, it propagates token 1 to the \textit{flag}. However, this will not work. In Fig. 3.12c al-
though the loop block has two loop headers, the irregular loop header $\text{Loopheader}_2$ gets the chance of being executed only for the first iteration. Assuming that the CFG enter into the loop from this irregular loop header, which results in such execution path $\text{Looppred}_2 \rightarrow \text{Loopheader}_2 \rightarrow \text{BBmerge} \rightarrow \text{Loopexit}_1$. Start from $\text{Loopexit}_1$, if the loop continues, then it jumps to the regular loop header $\text{Loopheader}_1$ instead of $\text{Loopheader}_2$. That means, in our solution shown in Fig. 3.12d $\text{Loopheader}_2$ gets the chance of being executed (i.e., $\text{flag} = 1$) only during the first execution of the loop; in the later iteration of the loop, $\text{Loopheader}_1$ should always be executed (i.e., $\text{flag} = 0$). Fig. 3.12e shows the synthesis circuit for $\text{flag}$, which has two $\text{MERGE}$. The $\text{MERGE}_1$ selects from tokens 0 and 1 which come from $\text{Looppred}_1$ and $\text{Looppred}_2$ respectively, and sends the result to $\text{MERGE}_0$, which is controlled by token $\text{loopVal}$. However, $\text{loopVal}$ connects to the control port of $\text{MERGE}_0$ through a $\text{INIT}$ operator with an initial token 1 inside. Therefore, $\text{MERGE}_0$ will select the result from $\text{MERGE}_1$ during the first iteration of the loop, and always select token 0 (i.e., $\text{Loopheader}_1$ is always executed) if the loop continues (i.e., $\text{loopVal} = 0$). When the loop finishes, $\text{loopVal} = 1$, which is stored at the $\text{INIT}$ operator, so the circuit is restored to the initial state after the execution of the loop.

**Summary.** A canonical CFG—an assumption implicit in previous work like [93]—requires that each if statement and loop statement has exactly one exit block. Unfortunately, it is easy to write software programs that violate this requirement. Examples of violations include loop statements that include a break, or a return inside any if or loop statement, both of which are common programming patterns. With the method proposed above, Fluid can process arbitrary goto-free programs.

### 3.4 Alternative Design

Section 3.3 shows the circuit design methodology, which uses the deterministic dataflow elements: $\text{MERGE}$ and $\text{SPLIT}$, to encode the software control flow information into the
circuit. The major design complexity comes from the control token generation (Section 3.3.2, Section 3.3.4, Section 3.3.5). There are two alternative designs that simplify the design complexity, and we study them in this section.

![Diagram of MIXER and ARBITER](image)

Figure 3.13: Non-deterministic dataflow elements

### 3.4.1 MIXER Design

A commonly used non-deterministic dataflow element is the uncontrolled **MERGE**, which is also called a **MIXER**. A **MIXER** shown in Fig. 3.13 is similar to a **MERGE**, except it does not have a control token port. A **MIXER** waits for an input token to arrive on any of its data ports, and propagates the first received input to the output port. If multiple input tokens arrive at the input of a **MIXER**, the output is non-deterministic; hence, dataflow graphs that use **MIXERS** often impose a mutual exclusion constraint on input token arrival so as to preserve deterministic execution.

If we use **MIXERS** in our design, then we don’t need to generate the control tokens. For the example in Fig. 3.8a, token \( y \) in \( B_5 \) receives multiple definitions (\( y_0, y_1 \) and \( y_2 \)) from its predecessors, and we could synthesize **MIXER** tree to select the right definition. As a result, the whole if-block (consisting of \( B_1, B_2, B_3, B_4, B_5 \)) has to be executed in a blocking manner. Consider the case where the CDFG in Fig. 3.8a is executed twice and generates \( y_0 \) and \( y_2 \) respectively. The **MIXER** operator requires mutual exclusiveness on the input ports, so \( y_2 \) could not be generated before \( y_0 \) is generated. Therefore, all of the branching sub-CDFG are mapped to unpipelined circuits.
3.4.2 ARBITER Design

Another non-deterministic dataflow element is ARBITER, shown in Fig. 3.13. It has two input data ports \textit{in1} and \textit{in2} and one data output port \textit{out}. It also has a control output port \textit{cOut}. The ARBITER would receive input tokens at the input ports and randomly propagate one of them to the output port, and the control output \textit{cOut} would specify which input data has been chosen. The output of ARBITER is non-deterministic, so dataflow graphs with ARBITERS often impose in-order constraints on input token arrival for deterministic execution.

For the example in Fig. 3.8a, Fluid synthesizes the collection circuit (Fig. 3.8c) to conditionally generate \textit{y} from multiple definitions, which requires control tokens \textit{c1} and \textit{c2}. We could use the ARBITER to generate the desired control tokens. However, the ARBITERS require in-order execution of the input tokens, so \textit{y0}, \textit{y1} and \textit{y2} have to be generated in the same order. To achieve that, the if-block (consisting of \textit{B1}, \textit{B2}, \textit{B3}, \textit{B4}, \textit{B5}) has to be non-pipelined.

\begin{summary}
The C programs require deterministic execution in nature. When the HLS engine maps them to hardware, it could reduce the design complexity by using the non-deterministic hardware operators. However, the HLS engine has to synthesize (partially) unpipelined circuits to get the deterministic results.
\end{summary}

3.5 Dataflow Graph Optimizations

Fluid converts the optimized LLVM IR into STF form, which is essentially a dataflow graph. In this section, we focus on optimizing the dataflow graph.

3.5.1 Operator Clustering

LLVM encodes expressions into three-address IR instructions, and Fluid maps each of
them into a dataflow operator, which is an independent pipelined process. For complex
expressions, Fluid generates many dataflow operators and misses opportunities for logic
optimizations across expressions. Therefore, it is desirable to group them together.

```c
1 int s1 = a + b;
2 int s2 = c + d;
3 int s3 = e * f;
4 int s4 = g * h;
5 int s5 = s1 * s2;
6 int s6 = s3 * s4;
7 int s7 = s5 + s6;
```

Fig. 3.14 (a) shows an example code for a chain of arithmetic operations, and Fig. 3.14
(b) shows the corresponding IR instructions generated by LLVM. For example, LLVM
maps

```c
1 int s1 = a + b;
```

into the IR instruction

```c
1 %9 = add nsw i32 %1, %2
```

The instruction specifies the *add* operation as well as the data types *i32* for both operands
and the result. The values (for operands and the result) are stored in the virtual registers
such as %1, %2, %9. Fluid will synthesize a *FUNC* operator to perform the *add* compu-
tation, and the input (output) channels of the operator carry the operands (result) of the
operator.

Fig. 3.14 (c) shows the vanilla dataflow graph Fluid generated directly from the LLVM
IR (Fig. 3.14 (b)). It contains four multipliers and three adders, as is specified by the

![Figure 3.14: Operator clustering example](image)
example code (Fig. 3.14 (a)). However, these operations are purely combinational logic, and Fluid groups these operations together and applies logic optimizations to generate a faster and smaller circuit (Fig. 3.14 (d)).

However, we could not naively cluster operations together by tracking the def-uses of each variable in the vanilla dataflow graph. The CFG of the source program divides the segments of codes into different control zones, and similarly the control nodes (i.e., \texttt{MERGE} and \texttt{SPLIT}) divide the whole graph into distinct control regions as well, and operator clustering is only applicable to nodes within the same control region.

To identify the clustering opportunities, we assign colors to each graph edge based on the condition it is activated. The in/out edges to a \texttt{FUNC} or a \texttt{COPY} node have the same color. The guard and input data edges of a \texttt{SPLIT}, or the guard and output edges of a \texttt{MERGE}, are also assigned the same color. The \texttt{BUF} and \texttt{INIT} nodes would generate a new color. The \texttt{SOURCE} operator only has an output edge, so it generates the initial color. The \texttt{SINK} operator only has an input edge, so it does not change any color, and can be clustered together with its predecessor. After coloring the dataflow graph as specified, the \texttt{FUNC} nodes whose output edges have the same color can now be safely clustered.

### 3.5.2 MERGE and SPLIT Tree Flattening

In Section 3.3 Fluid synthesizes \texttt{MERGES} and \texttt{SPLITS} for each unique condition variable in the collection and delivery conditions, potentially generating a tree of 2-way \texttt{MERGE}s and \texttt{SPLIT}s. Fluid further flattens them into the N-way \texttt{MERGE} and \texttt{SPLIT}, which reduce the delay, area and energy consumption.

To perform this transformation, we need to create a new control token using the control tokens for the 2-way \texttt{MERGE}s or \texttt{SPLIT}s. For a \texttt{SPLIT} tree, we mark the top \texttt{SPLIT} as "parent" node, and all the \texttt{SPLIT}s below as "child" node. It is required that the control tokens to the child node should be generated by a \texttt{SPLIT} whose input has the same color as the parent node’s control token, or by a function of such \texttt{SPLIT}s.
Fig. 3.15 shows a SPLIT tree which consists of SPLIT0 (parent node) and SPLIT1 (child node). For SPLIT0, its input data and control edges are colored grey, its left output edge (i.e., LOut0) is colored red, and its right output edge (i.e., ROut0) is colored blue. Since ROut0 is also the input data edge to SPLIT1, its control edge is colored blue as well. Lastly, we color the left (right) output edge of SPLIT1 as green (orange).

Let’s focus on CTRL1 (the control edge of SPLIT1), which has the same color (i.e., within the same control zone) with ROut0. Therefore, CTRL1 either directly comes from ROut0, or comes from another SPLIT2 whose input data and control edges are also colored gray (there could be FUNC blocks on the way). Fig. 3.15a shows the latter case. As shown in Fig. 3.15b, Fluid flattens SPLIT0 and SPLIT1 into a 3-way SPLIT3, and the input to SPLIT3 is the same as the input to SPLIT0. Fluid synthesizes a FUNC1 block to calculate the new control token, whose inputs are the control token of SPLIT0 and the input token of SPLIT2. If the outputs of SPLIT2 are not used elsewhere, Fluid deletes SPLIT2.

3.6 Evaluation

3.6.1 Control Circuit Synthesis

Each dataflow graph component is translated into a unique pipeline stage, and the data transfers between pipelined stages use the bundled data protocol [70]. The control for each pipelined stage uses micro-pipelines [91].

Fig. 3.16 shows a standard bundled data circuit template that we use in our evaluations.
The control path is the upper part in bold lines, and the data path is within the dashed boxes. The stage logic implements the function in the dataflow node, and the control circuit implements the four-phase handshake using a Muller C-element ($C$). When the input token is ready ($in.\text{rdy}$ signal is high), and the successor stage is empty ($out.\text{ack}$ is low), $C$’s output signal $s$ becomes high, which triggers data capture using a pulse generator $G$ and latch, and then the execution of the stage logic. When the output token is ready, the $out.\text{rdy}$ signal is set to high. After the next stage captures this data, it will set the acknowledged signal $out.\text{ack}$ to high, allowing the current stage to reset. Delay lines (15% slower than the worst-case delay of the stage logic) are added to ensure successful data capturing and processing.

### 3.6.2 Simulation Methodology

In order to simulate and measure the performance of synthesized asynchronous circuits, we built a discrete-event simulator that can simulate the execution of the bundled data circuits in Fig. 3.16. Each pipelined process fetches data from the predecessors and sends out results to its successors, and the simulator simulates the 4-phase handshake for process communication. Performance numbers for different circuit components are extracted using commercial tools, and used to annotate the discrete event simulator. Specifically, we use HSPICE to simulate the control circuit in a 28nm process technology. For the stage datapath logic (combinational), we used commercial logic synthesis tools and a commercial

![Figure 3.16: Bundled-data circuit template.](image)
28nm standard cell library to determine performance/power/area. The delay of each stage is the sum of the delay of the control circuit and the stage logic (as shown in Fig. 3.16). Synchronous results were obtained using the same cell library and same commercial logic synthesis tool.

3.6.3 Experimental Setup

**benchmark.** We synthesize the following microbenchmarks:

1. *arith*, which performs the same arithmetic computation as the example code in Fig. 3.14, i.e., $y = (x_0 + x_1) \times (x_2 + x_3) + (x_4 \times x_5) \times (x_6 \times x_7)$.

2. *if*, which is the same if statement as the example code in Fig. 1.3. The true branch does the addition and the false branch does division. The true branch will be triggered during runtime.

3. *for0*, which has a single loop which sums the integers from 0 to 9.

4. *for1*, which has two-layer nested loops. In the outer layer, the integer counter iterates over 0 to 9; in the inner layer, the integer counter iterates over 0 to 1, and increases the sum by 1 for every iteration.

5. *if-loop* which has an if statement: the true branch has a one-layer loop (count from 0 to 9) and the false branch which does the division. The true branch will be triggered during runtime.

We also extract five kernel functions from five applications which are mostly taken from an HLS benchmark suite [7] or used in synchronous ASIC synthesis benchmarking [76]:

1. *differential*, a differential equation solver [76].

2. *adpcm-u*, the uppol2 function from adpcm [7].
3. \textit{dfadd-a}. The add function from dfadd \cite{7}.

4. \textit{gsm-d}. The \texttt{gsm\_div} function from gsm \cite{7};

5. \textit{mpeg-d}. The decode function from mpeg \cite{7}.

**Comparison.** We compare our tool with \textit{LegUp} v4.0 which is a commonly used academic HLS tool, and two commercial HLS tools \textit{Commercial 1} and \textit{Commercial 2}. Furthermore, we have \textit{Fluid} (vanilla version of Fluid) and \textit{Fluid-opt} (Fluid with dataflow optimizations).

**Metrics.** We use the following performance metrics: Delay (\textit{ps}), Area (\textit{\mu m^2}), Energy (\textit{pJ}), LeakPower\textit{(nW)} and Throughput (\textit{MHz}). In order to get the throughput, we run each benchmark application for 20 times and record the total delay \(D(\text{ps})\). The throughput is calculated as \(\frac{20 \times 10^6}{D(\text{ps})} \text{(MHz)}\).

Fluid will directly extract the performance numbers from the annotated circuit simulator. \textit{LegUp}, \textit{Commercial 1} and \textit{Commercial 2} all generate RTL designs from C programs. We use commercial tools to do logic synthesis and optimizations, and measure the delay, area, dynamic power and leak power of the synthesized circuits. In order to measure their energy, we use the circuit’s total power (dynamic power + leak power) multiplying its delay.

We run each benchmark twenty times (the input data are provided by each benchmark, and there are dozens of them to cover different cases) and use averages across the runs to report benchmark statistics. We use the same methodology to collect results for both Fluid and the other HLS tools.

### 3.6.4 Experimental Results

Table 3.1 shows the performance of our system, and Table 3.2 shows other tools. Fig. 3.17 and Fig. 3.18 show the performance breakdown for all of the tools.

Each table has three sections: the first shows the performance of HLS benchmarks, and the second shows the performance of microbenchmarks. To summarize across benchmarks,
Table 3.1: Fluid and Fluid-opt Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Fluid</th>
<th></th>
<th></th>
<th></th>
<th>Fluid-opt</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm-u</td>
<td>11104</td>
<td>33814</td>
<td>84</td>
<td>10331</td>
<td>367</td>
<td>4588</td>
<td>7381</td>
<td>10</td>
</tr>
<tr>
<td>dfadd-a</td>
<td>9445</td>
<td>132021</td>
<td>31</td>
<td>35081</td>
<td>212</td>
<td>7415</td>
<td>64258</td>
<td>8</td>
</tr>
<tr>
<td>differential</td>
<td>226516</td>
<td>19051</td>
<td>1545</td>
<td>6217</td>
<td>4</td>
<td>16581</td>
<td>11180</td>
<td>1007</td>
</tr>
<tr>
<td>gsm-d</td>
<td>41378</td>
<td>6301</td>
<td>95</td>
<td>1721</td>
<td>25</td>
<td>26861</td>
<td>5274</td>
<td>51</td>
</tr>
<tr>
<td>mpeng-d</td>
<td>15844</td>
<td>22233</td>
<td>24</td>
<td>7147</td>
<td>99</td>
<td>7523</td>
<td>7285</td>
<td>5</td>
</tr>
<tr>
<td>arith</td>
<td>4378</td>
<td>10872</td>
<td>20</td>
<td>3634</td>
<td>563</td>
<td>3548</td>
<td>9268</td>
<td>29</td>
</tr>
<tr>
<td>if</td>
<td>1858</td>
<td>8506</td>
<td>2</td>
<td>2926</td>
<td>854</td>
<td>1566</td>
<td>1650</td>
<td>1</td>
</tr>
<tr>
<td>for0</td>
<td>19680</td>
<td>2596</td>
<td>55</td>
<td>755</td>
<td>52</td>
<td>13628</td>
<td>2080</td>
<td>18</td>
</tr>
<tr>
<td>for1</td>
<td>40079</td>
<td>4927</td>
<td>175</td>
<td>1410</td>
<td>26</td>
<td>30843</td>
<td>4356</td>
<td>61</td>
</tr>
<tr>
<td>if-loop</td>
<td>19946</td>
<td>10428</td>
<td>56</td>
<td>3469</td>
<td>52</td>
<td>14458</td>
<td>3410</td>
<td>18</td>
</tr>
<tr>
<td>Ratio1</td>
<td>1.03</td>
<td>2.68</td>
<td>0.41</td>
<td>2.35</td>
<td>1.63</td>
<td>0.61</td>
<td>1.19</td>
<td>0.12</td>
</tr>
<tr>
<td>Ratio2</td>
<td>0.68</td>
<td>0.73</td>
<td>0.21</td>
<td>0.74</td>
<td>1.99</td>
<td>0.52</td>
<td>0.38</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Table 3.2: Performance of LegUp 4.0 and two commercial HLS tools

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Commercial 1</th>
<th></th>
<th></th>
<th></th>
<th>Commercial 2</th>
<th></th>
<th></th>
<th></th>
<th>Legup</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm-u</td>
<td>6000</td>
<td>9846</td>
<td>81</td>
<td>4616</td>
<td>167</td>
<td>3750</td>
<td>6825</td>
<td>22</td>
<td>2123</td>
<td>267</td>
<td>6000</td>
<td>8013</td>
<td>66</td>
</tr>
<tr>
<td>dfadd-a</td>
<td>6000</td>
<td>8780</td>
<td>55</td>
<td>3282</td>
<td>167</td>
<td>6250</td>
<td>12154</td>
<td>54</td>
<td>3241</td>
<td>160</td>
<td>9000</td>
<td>10941</td>
<td>77</td>
</tr>
<tr>
<td>differential</td>
<td>130000</td>
<td>8771</td>
<td>1861</td>
<td>4208</td>
<td>8</td>
<td>217778</td>
<td>25491</td>
<td>6732</td>
<td>8907</td>
<td>5</td>
<td>203333</td>
<td>15948</td>
<td>3451</td>
</tr>
<tr>
<td>gsm-d</td>
<td>20667</td>
<td>2349</td>
<td>55</td>
<td>776</td>
<td>48</td>
<td>23333</td>
<td>9502</td>
<td>175</td>
<td>2813</td>
<td>43</td>
<td>28750</td>
<td>9682</td>
<td>176</td>
</tr>
<tr>
<td>mpeng-d</td>
<td>6000</td>
<td>4096</td>
<td>28</td>
<td>1651</td>
<td>167</td>
<td>45000</td>
<td>11305</td>
<td>470</td>
<td>3587</td>
<td>22</td>
<td>42222</td>
<td>6327</td>
<td>267</td>
</tr>
<tr>
<td>arith</td>
<td>5500</td>
<td>8584</td>
<td>83</td>
<td>4512</td>
<td>182</td>
<td>6250</td>
<td>13026</td>
<td>88</td>
<td>4106</td>
<td>160</td>
<td>11250</td>
<td>13156</td>
<td>204</td>
</tr>
<tr>
<td>if</td>
<td>4000</td>
<td>3128</td>
<td>25</td>
<td>1273</td>
<td>250</td>
<td>3750</td>
<td>10482</td>
<td>41</td>
<td>3006</td>
<td>267</td>
<td>10000</td>
<td>15342</td>
<td>153</td>
</tr>
<tr>
<td>for0</td>
<td>10500</td>
<td>2082</td>
<td>22</td>
<td>800</td>
<td>95</td>
<td>6250</td>
<td>4396</td>
<td>25</td>
<td>1376</td>
<td>160</td>
<td>16667</td>
<td>3974</td>
<td>47</td>
</tr>
<tr>
<td>for1</td>
<td>25600</td>
<td>3447</td>
<td>149</td>
<td>1421</td>
<td>39</td>
<td>6250</td>
<td>3949</td>
<td>23</td>
<td>1245</td>
<td>160</td>
<td>35000</td>
<td>4065</td>
<td>127</td>
</tr>
<tr>
<td>if-loop</td>
<td>11000</td>
<td>4699</td>
<td>86</td>
<td>1960</td>
<td>91</td>
<td>7500</td>
<td>14034</td>
<td>105</td>
<td>4119</td>
<td>133</td>
<td>13750</td>
<td>18772</td>
<td>257</td>
</tr>
<tr>
<td>Ratio1</td>
<td>0.53</td>
<td>0.61</td>
<td>0.43</td>
<td>0.73</td>
<td>1.87</td>
<td>0.83</td>
<td>1.22</td>
<td>0.95</td>
<td>1.10</td>
<td>1.20</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ratio2</td>
<td>0.59</td>
<td>0.43</td>
<td>0.42</td>
<td>0.59</td>
<td>1.69</td>
<td>0.38</td>
<td>0.89</td>
<td>0.34</td>
<td>0.86</td>
<td>2.65</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
we use the geometric mean of the normalized performance compared to LegUp; Ratio1 corresponds to the HLS benchmarks, and Ratio2 corresponds to microbenchmarks.

Figure 3.17: Performance breakdown I

**Delay.** Fig. 3.17a shows the delay comparison. Fluid has a longer delay for two reasons: 1)
it fails to do logic optimizations for operator clusters; 2) it synthesizes MERGES and SPLITs which contribute to the extra delay. Fluid-opt can avoid the extra delay from reason 1. However, neither of them performs well for for0, for1 or if-loop. These three benchmarks are basically simple loops, so Fluid and Fluid-opt spend quite a lot of time on the MERGE trees and the SPLIT trees.

For if benchmark, Fluid and Fluid-opt perform well because they generate asynchronous circuits whose actual delay depends on the activated processes during runtime (i.e., the addition). The other tools, however, are limited by the worst-case scenario (e.g. division) when generating the global state machine for the circuits.

Fluid-opt reduces delay by 1.64X and 1.92X for HLS benchmarks and microbenchmarks respectively.

**Area and Leakage.**

Compared with synchronous circuits, the asynchronous circuits need to pay extra cost for the control circuitry (Section 3.6.1) as well as the control elements MERGES and SPLITs (Section 3.3). On the other hand, the asynchronous circuit does not require a global state machine to control the circuit execution, which saves area. Fluid-opt applies operator clustering algorithm, which could leverage the logic optimizations opportunities for combinational computations, which could further reduce the area. Note that Legup, Commercial 1 and Commercial 2 all use commercial tools to do logic optimizations on their generated circuits, so the operator clustering is also applied on their outputs by the commercial tools.

Fig. 3.17b shows the normalized area performance. Note that we truncate the y-axis at y = 2 position for illustration purposes. Fluid and Fluid-opt performs badly in dfadd-a, because this benchmark has many-layer nested if statements and loop statements, and our system generates huge MERGE trees and SPLIT trees to collect and deliver data tokens. But for other benchmarks, our system can achieve similar or even better performance in terms of area.

It is worth noting that Fluid-opt improves a lot in area compared with Fluid. It is
because Fluid-opt could cluster multiple operators into one and do logic optimizations to it as well as flatten the *merge* and *split* trees, which reduces area by a large portion.

Fluid-opt increases HLS benchmark area by $1.19X$, while reducing it by $2.63X$ for microbenchmarks.

Leakage power results are qualitatively similar to those for area.

![Energy Comparison](image1)

**Figure 3.18a: Energy comparison**

![Throughput Comparison](image2)

**Figure 3.18b: Throughput comparison**

Energy. Compared with synchronous design, asynchronous circuits only activate the processes that receive the input data, which could save energy. On the other hand, asynchronous circuits need to pay extra cost for the control circuits for each process to communicate with other processes.

Fig. 3.18a shows the normalized energy performance. Fluid and Fluid-opt achieve bet-
ter energy performance in most benchmarks. However, in \textit{for0} and \textit{for1}, Fluid have higher energy consumption, since it needs to synthesize multiple \textit{merge} and \textit{split} trees for the data tokens. Fluid-opt could further improve energy performance compared with Fluid by doing logic optimizations for the combinational operator clusters.

Table 3.1 shows that Fluid-opt reduces energy by 8.33$X$ and 9.09$X$ for HLS benchmarks and microbenchmarks compared to \textit{LegUp}.

**Throughput.** Fluid synthesizes highly pipelined circuits, which could effectively increase the circuit throughput. As shown in Fig. 3.18b, Fluid and Fluid-opt has higher throughput than other tools for most of the benchmarks.

Note that the operator clustering optimization by Fluid-opt has two-sided effects on throughput: on one hand, it could reduce the latency of operator clusters (thus the whole circuit), which could increase the throughput; on the other hand, creating a large combinational operator out of multiple smaller combinational operators would reduce the pipeline stages and thus reduce the throughput. Fluid has higher throughput than Fluid-opt for \textit{arith} benchmark, but it has lower throughputs than Fluid-opt in other benchmarks, so the operator clustering optimization in general is beneficial for throughput improvement.

Fluid and Fluid-opt do not perform well for loop benchmarks (\textit{for0} and \textit{for1}). This is because these two benchmarks have quite simple loop bodies, and their throughputs are limited by the loop dependencies. This work does not optimize for loops specifically, so Fluid and Fluid-opt have mediocre performance for benchmarks whose throughputs are mainly bounded by loop dependencies.

Table 3.1 shows that Fluid-opt has higher throughput than Fluid for most benchmarks. Fluid-opt increases the throughput by 2.5$X$ for HLS benchmarks, and 2.54$X$ for microbenchmarks compared to \textit{LegUp}.

**Comparison with CPU.** We studied the performance comparison between Fluid-opt, which generates the ASIC implementations for the benchmarks, and the CPU implementations by directly running the benchmarks on commodity servers. We use \textit{GCC 10.2.0} as the C
compiler, and we turn on the optimization level -O2.

Table 3.3: Fluid-opt vs CPU implementation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CPU runtime (ps)</th>
<th>Fluid runtime (ps)</th>
<th>Speedup (X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm-uppol2</td>
<td>166481</td>
<td>4588</td>
<td>36.29</td>
</tr>
<tr>
<td>dfadd</td>
<td>256971</td>
<td>7415</td>
<td>34.66</td>
</tr>
<tr>
<td>differential</td>
<td>256471</td>
<td>165814</td>
<td>1.55</td>
</tr>
<tr>
<td>gsm-div</td>
<td>288467</td>
<td>26861</td>
<td>10.74</td>
</tr>
<tr>
<td>mpeg-decode</td>
<td>167981</td>
<td>7523</td>
<td>22.33</td>
</tr>
<tr>
<td>arith</td>
<td>146983</td>
<td>3548</td>
<td>41.43</td>
</tr>
<tr>
<td>if</td>
<td>161482</td>
<td>1566</td>
<td>103.12</td>
</tr>
<tr>
<td>for0</td>
<td>158482</td>
<td>13628</td>
<td>11.63</td>
</tr>
<tr>
<td>for1</td>
<td>165981</td>
<td>30843</td>
<td>5.38</td>
</tr>
<tr>
<td>if-loop</td>
<td>174480</td>
<td>14458</td>
<td>12.07</td>
</tr>
</tbody>
</table>

Table 3.3 shows the runtime comparison for Fluid-opt and the CPU implementation. The ASIC implementation could achieve 30X – 100X speedup for most benchmarks. However, if the benchmarks mainly have loops (gsm-div, differential, for0, for1, if-loop), the ASIC implementations have limited speedup. For differential benchmark, it is essentially a loop whose body consists of multiple multiplications. Fluid could not pipeline the loop well due to the loop dependency, and it also generated expensive (in terms of delay) circuits for the multiplication.

**Overall performance.**

No HLS tool is consistently the best in all metrics across all benchmarks, and the key is to achieve good trade-off among the metrics. Therefore, Fig. 3.19 shows per-benchmark spider plots of normalized performance as well as the geometric mean of the normalized performance of HLS benchmarks, and Fig. 3.20 shows those for the micro benchmarks. We plot the inverse of the normalized throughput, so for all metrics lower is better. Fluid-opt achieved a good balance among the five metrics for most of the benchmarks. Note that Fluid by itself rarely compares favorably against commercial HLS tools, so the dataflow graph optimizations are an essential ingredient of the overall flow.
Figure 3.19: Normalized performance for HLS benchmarks: delay (D), area (A), energy (E), leakage power (L), and throughput-inv (1/T)
Figure 3.20: Normalized performance for micro benchmarks: delay (D), area (A), energy (E), leakage power (L), and throughput-inv (1/T)
3.7 Related Work

Asynchronous synthesis. [49, 27, 34] are based on syntax-directed translation of the syntax of a message-passing hardware description language (HDL) into an asynchronous circuit. [102] uses Petri-net based synthesis of timed circuits from a message-passing HDL. [74, 73] use scheduling analysis similar to synchronous HLS tools, and emit a HDL program that is mapped to asynchronous circuits via syntax-directed translation. Unlike these tools, Fluid directly generates dataflow graphs and circuits.

[53] proposes a source-to-source transformation with concurrency optimizations. [54] proposes a new scheduling algorithm for generating asynchronous design out of synchronous one by removing the discrete time assumption. [100, 93] synthesizes circuits in a data-driven manner using CFGs, targeting pipelined processes. [92] allows designers to explicitly express the data-flow. Fluid can create dataflow graphs from software programs for a larger class of CFGs compared to them.

[95, 33] compiles C programs to Pegasus [32] IR, which is later synthesized to the bundled data circuit. This work only uses conditional tokens to implement loops, while if-statements compute both branches and select the result. Fluid uses conditional execution for if-statements as well to save energy, and handles more complex CFGs as detailed in Section 3.3.

Dataflow HLS. Some synchronous HLS tools generate elastic dataflow circuits. [43, 94, 57] directly map software programs to dataflow circuits. Fluid can handle more complex program control structures. Some other works propose domain-specific languages [86, 28, 4, 84, 39, 21] and special directives/pragmas [48, 67, 80] to help the synthesis. Fluid, however, does not require any changes to the source code or language-level support.

[43] propose a protocol for implementing elastic communication channels for synchronous systems. [94] uses syntax-directed translation to map functional programs into dataflow circuits leveraging the functional programming model. [57] generates elastic cir-
cuits, and optimizes for memory access. CAPH \cite{86} is a domain-specific language for streaming applications. \cite{84} extends \cite{86} for CGRA. CAL \cite{4} is designed for dataflow programming with actors, and OpenDF \cite{28} converts CAL code to HDL. \cite{39} generates elastic circuits for stencil computation. Fluid uses standard C as input, includes optimizations for *splits/merges*, and handles complex *cfgs*.

\cite{48} uses directives to control array partitioning, inlining options, etc. for mapreduce programs. \cite{67} generates HLS directives to partition the program into different clock domains. \cite{80} uses LLVM to generate IR from C programs, applies optimizations such as vectorization and loop unrolling based on resource models. After that, these works \cite{48, 67, 80} rely on the commercial tools to generate the circuits. Fluid is orthogonal to these approaches and can serve as an alternative back-end for asynchronous circuit generation.

Without the complex *cfg* support in Fluid, *dfadd, adpcm, gsm* cannot be synthesized into the dataflow circuit. Fluid also has several dataflow optimizations, including that for *split* and *merge* nodes. Lastly, Fluid compares favorably with several HLS tools (including commercial tools).

### 3.8 Summary

We propose a new solution to dataflow circuit generation that can handle real applications with complex control structures. We also include dataflow optimization prior to generating the final asynchronous circuit, including optimizations that can handle conditional components. We compare our work against three HLS tools, and show that our work achieves improvements in terms of energy and throughput.
Chapter 4

PipeLink: A Pipelined Resource Sharing System for Dataflow High-level Synthesis

4.1 Introduction

In Chapter 3, we present the compilation stage of our tool that maps each function in high level programs (with complex control structures) into independent asynchronous dataflow circuits. There are two major implications of adopting the dataflow model of computation. First, the implementation is inherently pipelined. Second, the pipelined nature of the implementation necessitates additional complexity to manage any resource sharing across different regions of the dataflow graph. This is handled by the linking stage of our tool.

In this chapter, we present PipeLink: a pipelined resource sharing system for asynchronous dataflow circuits. PipeLink makes a number of contributions to dataflow HLS: (i) it includes a general pipelined resource sharing mechanism that supports pipelined access to a shared resource; (ii) it leverages this resource sharing mechanism to support both memory access as well as (non-inlined) function calls; (iii) it enables separate compilation of individual functions into dataflow graph fragments; (iv) it provides more general support for function pointers, enabling translations of programs that are unsupported even by some state-of-the-art commercial tools; (v) it implements a range of dataflow graph opti-
mizations that improve the quality of the final dataflow graph implementation. We evaluate our HLS approach against both academic and commercial HLS tools. Overall, PipeLink provides significant improvements in energy, latency, and throughput while paying an area penalty. Compared to commercial (academic) HLS tools, PipeLink results in 12X (20X) reduction in energy, 1.29X (1.64X) improvement in throughput, 1.27X (1.61X) improvement in latency at a cost of 2.4X (1.61X) increase in area.

4.2 Background and Related Work

Dataflow circuits consist of dataflow elements that communicate with their predecessors and successors through handshake protocols. The dataflow elements are introduced in Section 3.2.2.

Another commonly used dataflow element is the uncontrolled \textit{MERGE} (also called a \textit{MIXER}) as specified in Section 3.4.1. Previous work has shown how simple programs consisting of single functions can be translated to \textit{MIXER}-free dataflow graphs using the control dataflow graph (CDFG) of the program [93, 66]. In this approach, function calls are inlined prior to dataflow graph generation. In what follows, dataflow graphs are \textit{MIXER}-free unless explicitly specified.

4.2.1 The Resource Sharing Problem in Dataflow Circuits

Suppose $f$ is a single input, single output function that has been translated into a dataflow graph. Furthermore, consider the problem of dataflow graph generation for the following program fragment:

\begin{verbatim}
1     y = f(x); \\
2     ... \\
3     z = f(w);
\end{verbatim}
There are two call sites for $f$. PipeLink treats functions and memories as Resources, and it needs to synthesize a resource access system to handle the resource access requests.

Fig. 4.1 illustrates the resource access problem. The Resource block is accessed from multiple call sites $1, 2, \ldots, m$, and the call site $i$ has Sender $i$ to send the input token(s) to the Resource block, and Receiver $i$ to receive the result token(s) from the Resource block.

There are three ways of accessing the Resource block:

1. resource replication. In Fig. 4.1 (a), the Resource block is replicated for $m$ times for each call site $i$. This allows concurrent execution of the call sites at the cost of wasting resources. Furthermore, not all Resource blocks can be replicated naively, i.e., the shared memories.

2. blocking access. In Fig. 4.1 (b), $m$ senders (receivers) are trying to send (receive) input (result) tokens to the Resource block, and there are “Collection Circuit” to se-
select the appropriate input token to route to the function, and the “Delivery Circuit” to route the result of the function back to the corresponding receiver. Note that neither the “Collection Circuit” nor the “Delivery Circuit” has the control token, so they could not figure out the order in which the senders and receivers could interact with the Resource. To guarantee the correctness, the “Collection Circuit” and the “Delivery Circuit” work in a blocking manner.

3. pipelined access. In Fig. 4.1 (c), the same “Collection Circuit” and the “Delivery Circuit” are synthesized as the datapath to access the Resource. A pipelined resource sharing system is also synthesized, which generates the $ctrl$ tokens to allow different senders and receivers to access the Resource in a pipelined manner.

A lot of existing HLS tools [19, 3, 9, 35, 57, 6, 78, 98, 81, 72] use replication-based methods to handle the resource sharing problem. They map each function into a separate hardware module, and instantiate these modules at multiple places by their callers. If some resource cannot be replicated (i.e., shared memories), then the blocking access circuitry (Fig. 4.1 (b)) is synthesized. The “collection circuit” is constructed using $MIXERS$, which propagates any input token arriving at the data port to the output port without any control. Since $MIXERS$ have non-deterministic behaviors, it is required that the inputs to the $MIXER$ be mutually exclusive for function correctness. This is especially important if the function has side-effects (i.e., accessing global memories). This design only supports blocking access, otherwise the callers could access the Resource in an out-of-order manner, which is different from the program’s specifications and generates the wrong results.

Fig. 4.1 (c) shows the circuit template for pipelined resource access. It has a pipelined resource sharing system that generates $ctrl$ tokens to the collection and delivery circuit. Furthermore, the Resource block itself needs to be a pipelined circuit. Lastly, the “Collection Circuit” and the “Delivery Circuit” also need to be pipelined. Section 4.3 details our approach to the generation of control tokens for pipelined resource sharing.
4.2.2 Related Work

**Resource sharing in HLS.** [60] introduces the synchronous HLS, where each operator is scheduled into different clock cycles. If function units among operations of the same type are accessed at multiple places, the HLS tools will share them as long as the accesses happen in different cycles.

Starting from this, many works focus on discovering more sharing opportunities. [71] shares (sub)modules across module boundaries to save area. [31] hierarchically synthesize the circuits using the sub-designs, but the states of the concerned modules need to be pre-determined statically. [41] tries to extract common structures or patterns in the dataflow graph, so that different instances of the same pattern can share resources. Traditional resource sharing requires the delay info of the resource, but [64] shares resources whose delays are unbounded. It extends the scheduling formation to satisfy the timing and handshaking requirements.

These works focus on identifying more resource sharing opportunities, which are orthogonal to PipeLink.

**Function pipelining in statically scheduled HLS.** Statically scheduled HLS tools [5, 19, 2] map each function into an independent hardware module. If a function is invoked frequently, then the tools will explicitly introduce fixed length pipelines within the function, and incorporate this information into the scheduling algorithm. To achieve that, typically all the loops are unrolled and sub-functions are inlined. PipeLink, however, can synthesize a pipelined resource sharing block for each function without such constraints.

**Dataflow HLS.** Some synchronous HLS tools [43, 94, 57, 38] generate elastic dataflow circuits by directly mapping software programs to dataflow circuits. [43] propose a protocol for implementing elastic communication channels for synchronous systems. [94] uses syntax-directed translation to map functional programs into dataflow circuits leveraging the functional programming model. [57] uses dynamic scheduling to generate elastic circuits.
and optimizes for memory access while [38] combines static and dynamic scheduling to generate circuits composed of elastic and statically-scheduled components. These works use MIXERS to access a shared resource, and hence have to access shared resources in a blocking manner. PipeLink provides a different approach to the translation, guaranteeing deterministic execution via the use of MERGE elements.

**Asynchronous synthesis.** Asynchronous circuits are a common target for dataflow synthesis. [95, 33, 66] maps C programs into asynchronous circuits. [95, 33] only supports non-pipelined access to shared functions while [66] only allows each function to be called at most once. [100, 93, 50] map behavioral hardware description languages into asynchronous circuits. [50] only allows non-pipelined function calls, and [100, 93] only allows a single function call per loop iteration. PipeLink can map C programs to asynchronous circuits with generic, pipelined function calls.

### 4.3 Pipelined Resource Sharing

The dataflow graph for a function has input ports (for arguments) and output ports (for return values). If the same function is called from multiple program points, then the ports for the function must be accessed (i.e., shared) from multiple program points. To enable pipelined access to the shared function, a MERGE tree (the “collection circuit”) and SPLIT tree (the “delivery circuit”) are used to create private ports to each unique call site for the function, as illustrated in Figure 4.1(c), along with the appropriate control. In this section we describe how we generate the appropriate control tokens and the SPLIT/MERGE data routing logic in a systematic manner based on the structure of the program.

Consider the simple scenario illustrated in Fig. 4.2(a). If this is the complete program, then the two call sites to $f$ strictly alternate. Hence, a control token sequence of 0, 1, 0, 1, ... (to the pipelined collection and delivery circuits in Fig. 4.1(c)) correctly routes data tokens to/from the dataflow graph for the function. If, on the other hand, we have a
1 \ y_0 = f(x_0); \\
2 \ y_1 = f(x_1); \\
1 \ y_0 = f(x_0); \\
2 \ y_1 = f(x_1); \\
3 \ y_2 = f(x_2); \\
3 \ y_2 = f(x_2); \\
1 \ \textit{y}_0 = f(x_0); \\
2 \ \textit{y}_1 = f(x_1); \\
1 \ y_0 = f(x_0); \\
2 \ \textit{y}_1 = f(x_1); \\
3 \ y_2 = f(x_2); \\
3 \ y_2 = f(x_2); \\

a) Sequential code (base case) \\
b) Sequential code (general case) \\
c) if code \\
d) loop code \\

Figure 4.2: Example code for token network

scenario such as Fig. 4.2(c), then the second call to \( f \) is conditional. Hence, in this case we need to \textit{compute} the control token sequence for the collection and delivery circuits based on the run-time value of the condition \( c \). This means, unlike the simple case in Fig. 4.2(a) where the desired control sequence can be pre-determined during synthesis stage, the control sequence for the more complex case in Fig. 4.2(c) could only be decided during runtime.

To solve this problem, we introduce an auxiliary one-bit \textit{use-resource} token stream that determines how many times \( f \) is called (used) at run-time by a program fragment. The value is run-length encoded; if \( f \) is called three times, then the use-resource token sequence would be 1, 1, 1, 0 indicating three uses of \( f \) for the particular invocation of the program fragment. Note that different invocations of a program fragment may use \( f \) a different number of times, and so a different use-resource token sequence could be generated for different invocations of the same program fragment. Take the code in Fig. 4.2(c) as an example. During the first run, \( c \) is true and \( f \) is used twice, so the desired use-resource token sequence is 1, 1, 0. During the second run, however, \( c \) is false and \( f \) is only used once, then the use-resource token needs to be dynamically adjusted to 1, 0.

Since the programming languages usually have three types of control structures: \textit{sequential}, \textit{conditional} and \textit{iteration}, we talk about the use-resource token sequence generation under these cases.
Base case. The base case for a function call is a single call site. The one-bit use-resource sequence for this case is simply 1, 0, 1, 0, ... , since the function is called exactly once every time this call site is encountered. No additional SPLIT/MERGE logic is needed to route arguments to/from the function. Note that there is another base case when there is no call to the function (use-resource sequence 0, 0, ...), but we optimize this away during graph construction.

Sequential composition: SEQ. If we have two programs that are composed in a sequential fashion and both of those may include an invocation of f, then each program generates its own use-resource sequence. To combine the two programs:

- (a) We introduce a two-way MERGE and two-way SPLIT to route the arguments and return value of f to/from the dataflow graphs of the two programs.

- (b) We compute the use-resource sequence for the composed program from the sequences for the two sub-programs.

- (c) We generate a control token sequence for the two-way MERGE and SPLIT to appropriately route data tokens to/from the shared function from/to the two sub-programs; these data routing SPLIT and MERGE blocks are part of the datapath, as they route data values that are part of the computation of f.

For point (a), the MERGE and SPLIT trees can be constructed using the techniques in Section 3.3.
For point (b), the use-resource token streams are combined as follows: “1” tokens are read from the first use-resource token stream and propagated to the output; when a “0” is encountered (i.e., the first sub-program has finished using $f$), the circuit switches to the second use-resource token stream and continues forwarding “1”s to the output until a “0” is encountered (i.e., the second sub-program has also finished using $f$). At this point, a “0” is produced, and the circuit switches back to the first token stream. For example, the base sequential example in Fig. 4.2 (a) can be treated as a composition of the following two sub-programs:

\begin{verbatim}
  y0 = f(x0);  
\end{verbatim}

and

\begin{verbatim}
  y1 = f(x1);  
\end{verbatim}

For each run, the first sub-program would generate the following use-resource token stream: 1, 0, so the final use-resource token stream would generate a 1 and discards the 0. Then, the second sub-program would also generate the following use-resource token stream: 1, 0, and the final use-resource token stream would generate a 1 followed by 0. Then, it waits for the new use-resource token stream from the first sub-program. In conclusion, for each run of the code in Fig. 4.2 (a), the final use-resource token stream is 1 (from the first sub-program), 1 (from the second sub-program), 0 (both sub-programs have finished).

Now let’s consider the general sequential example in Fig. 4.2 (b), which can be treated as a composition of the following sub-programs:

\begin{verbatim}
  y0 = f(x0);  
\end{verbatim}

and

\begin{verbatim}
  y1 = f(x1);  
\end{verbatim}

\begin{verbatim}
  y2 = f(x2);  
\end{verbatim}
For each run, the first sub-program would generate the following use-resource token stream: 1, 0, and the second sub-program would generate the following use-resource token stream: 1, 1, 0. Therefore, the final use-resource token stream would be: 1, 1, 1, 0 for each run.

For point (c), control tokens for the \textit{split} and \textit{merge} for routing data tokens are generated as follows: for each “1” corresponding to the first token stream, a “0” token is generated for the datapath; for each “1” corresponding to the second token stream, a “1” token is generated for the datapath.

Fig. 4.3(a) and Fig. 4.3(b) illustrate how the use-resource tokens and control tokens are generated in a systematic fashion for example programs Fig. 4.2(a) and Fig. 4.2(b) respectively. The datapath control tokens are shown in red, and the use-resource tokens are shown in green. Assuming that the example codes are running once. In Fig. 4.3(a), \( t_{01} \) is 1, 1, 0 for each run, and \( dp_{01} \) is 0, 1 respectively, so \( x_0 \) and \( x_1 \) are sent to the \( f \) sequentially. In Fig. 4.3(b), \( t_{12} \) is 1, 1, 0, and \( dp_{12} \) is 0, 1, so \( x_1 \) and \( x_2 \) are sent to the output port of \( \text{merge}_1 \) sequentially. \( t_{012} \) is 1, 1, 1, 0 and \( dp_{012} \) is 0, 1, 1 accordingly, so \( x_0 \) and the two outputs from \( \text{merge}_1 \) (i.e., \( x_1 \) and \( x_2 \)) are sent to \( f \) sequentially.

**Conditional composition: \textit{IF} \textit{c}.** If \( f \) is called in a conditional fashion where \( c \) is the condition, then the value of the condition can be used to generate the use-token sequence. If we have a program fragment that is of the form

```
  if (c) {
    S;
  } else {
    T;
  }
```

then

- (i) if \( c \) is false, code segment \( T \) is executed, and we propagate the 1 in use-resource
token sequence from $T$ to the final use-resource token stream until a “0” is encountered;

- (ii) if $c$ is true, code segment $S$ is executed, and we propagate the 1 in use-resource token sequence from $S$ to the final use-resource token stream until a “0” is encountered;

A $\text{SPLIT/MERGE}$ pair is generated as in the $\text{SEQ}$ case, along with the accompanying control tokens that are either $1\ldots 1_N$ or $0\ldots 0_N$ based on whether $S$ or $T$ is executed. $N$ depends on the number of “1”s in the use-resource token sequence from $S$ or $T$.

Fig. 4.2(c) shows the case where $f$ is invoked twice: $f^1$ (Line 3) executed in if block $IF_c$, and $f^0$ (Line 1) and $IF_c$ are executed sequentially. Fig. 4.3(c) shows the synthesized circuits. Since $f^1$ is conditionally invoked, PipeLink creates $\text{MERGE}_1$ and $\text{SPLIT}_1$ to conditionally send argument $x_1$ to $f$ and collect result for $y_1$, which are controlled by the $IF_c$ network. In general $IF_c$ network takes in three inputs: if condition token $c$, and two input use-resource tokens corresponding to the function invocations in both branches. (Note that we have omitted the constant “0” use-resource stream as an input to $IF_c$, since $f$ is only used in the true branch.)

**Iterative composition: $\text{LOOP}_c$.** If $f$ is called in a loop where $c$ is the loop condition, then the value of the condition can be used to generate the use-token sequence. If we have a program fragment that is of the form

```plaintext
1  while (c) {
2    $S$;
3  }
```

suppose the loop runs for $m$ iterations (i.e., the code segment $S$ is executed for $m$ times), then the use-resource token stream for the body of the loop is propagated to the output in the form of $1 \ldots 1_{N_1+N_2+\ldots+N_m} 0$. During the $i$-th iteration, $S$ executes $f$ for $N_i$ times,
generating the use-resource token stream $1 \ldots 1 0$. We propagate the "$1" to the final output resource-resource token stream and discard the "$0" unless it is the end of the loop.

Note that in different iterations, $S$ could execute $f$ for different times. For example, consider the following loop example:

```
1 while (c) {
2   c2=...
3   if (c2) {
4       ... call f ...
5   }
6 }
```

$f$ is conditionally executed for each loop iteration.

Since the loop has a single body, no \textit{split/merge} or control token generation is needed. Fig. 4.2(d) shows the case where $f$ is invoked twice: $f^0$ (Line 2) executed in loop block $\text{loop}_C$, and $f^1$ (Line 4) executed sequentially after $\text{loop}_C$. Fig. 4.3(d) shows the synthesized circuits. $\text{merge}_0$ and $\text{split}_0$ are synthesized for the sequential execution. Note that $x_0$ ($y_0$) is directly connected to the $\text{merge}_0$ ($\text{split}_0$). This is because $x_0$ is already conditionally generated under the loop condition $c$ (in Section 3.3.3).

**Summary.** There are three building blocks used to compose use-resource token networks: $\text{seq}$, $\text{if}_c$ and $\text{loop}_c$. We use these blocks to systematically construct the final use-resource token network for a complete program by induction on the structure of the program. Each basic type of token network receives use-resource tokens from base-case generators or other sub-token networks, and generates one use-resource token. In addition, $\text{seq}$ and $\text{if}_c$ also generate datapath control tokens along with $\text{split/merge}$ data routing logic. The use-resource token network operates in parallel with the datapath. To reduce stalls in the use-resource network, PipeLink inserts FIFO buffers at the delivery circuit end (shown as yellow blocks in Fig. 4.3). As a result, new datapath control tokens can be generated (thus
enabling new invocations to start) without waiting for the previous invocation to finish. The final unused use-resource tokens are connected to a \textit{sink} operator.

For a shared function call to \( f \) from \( g \), we denote the use-resource \textit{token network} by \( TN(g,f) \), the \textit{merge tree} collection network datapath by \( COL(g,f) \), and the \textit{split tree} delivery network datapath by \( DEL(g,f) \). The three basic token networks are pre-designed template circuits, whose details can be found at Appendix A.

\section*{4.4 Static Dataflow Synthesis}

Like other HLS tools, PipeLink does not support recursive function calls. Hence, we can build an acyclic call graph for the complete program with the top-level function as the root. In addition, we augment the call graph with the names of arrays (i.e. memory) and accesses to memory. We call this the \textit{extended} call graph.

Fig. 4.4 shows a program and its extended call graph. \( m \) is the function to be translated into a dataflow circuit, and it directly accesses array \( A \) and calls \( g \). \( g \) calls \( f \) and \( h \). These relationships are captured by the extended call graph (Fig. 4.4).

PipeLink constructs the whole program dataflow graph in two stages that we view as \textit{compilation} and \textit{linking}. During the compilation stage, each function is mapped to a dataflow circuit using existing techniques \cite{93,66}. However, any resource access request (to memories and other functions) remains unresolved as “dangling” dataflow connections (i.e. ports). During linking, PipeLink collects the resource access requirements across the whole program to construct the pipelined linking circuit and complete the dataflow graph.

For the purposes of dataflow graph construction, a memory can be viewed in the same way as a function. A memory has input ports for address, read/write command, and data (used for write operations), and an output port that produces data (used for read operations). Hence, PipeLink treats memories and ordinary functions in a uniform way. In this chapter, we use “function” to denote both regular functions and memory banks unless otherwise
Figure 4.4: An example program with multiple function calls, and the corresponding extended call graph.
specified. Memory blocks in the extended call graph can come from explicitly declared arrays (local or global), as well as dynamically allocated arrays (e.g., `malloc`).

Each non-root node $f_1$ in the extended call graph has an immediate dominator $f_2$; we say that $f_1$ is the guest function for $f_2$, and $f_2$ is the host function of $f_1$ (i.e., $f_1$ hosts $f_2$). In PipeLink, the host of a function $f$ is responsible for coordinating the access to $f$’s input and output ports.

Hence, if $f_1$ calls $f_2$ which it does not host, then it cannot directly access the I/O ports of $f_2$; instead, $f_1$ will create an internal proxy function $f_1f_2$, which will take over its request to $f_2$ for $f_1$.

Considering the example in Fig. 4.4(a), which has four regular functions $m, g, h, f$ and one memory bank $A$. During the compilation stage, five circuit objects are generated for each of them, and the code snippet shows the resource access requests that remain unresolved. During linking stage, PipeLink constructs the extended call graph, where $m$ hosts $g$ and $A$, and $g$ hosts $f$ and $h$. $h$ tries to access $f$ and $A$ which it does not host, so it creates two proxy functions $hf$ and $hA$ internally. Although $g$ does not explicitly call $A$, it needs to handle the redirected request from $h$ (more specifically, $hA$) to $A$, therefore it creates a proxy function $gA$ to access $A$ as well.

If $f_1$ calls $f_2$ either directly (e.g., $g$ calls $f$) or indirectly through another function (e.g., $g$ calls $A$), there are two cases: (i) hosted calls, when $f_1$ hosts $f_2$; or (ii) nested function calls, where $f_1$ is not the host function of $f_2$.

1. $f_1$ hosts $f_2$. In this case, $f_2$ is instantiated by $f_1$, which means $f_1$ has direct access to $f_2$’s I/O ports. PipeLink proposes Token Network Design (Section 4.4.1) to allow pipelined access to $f_2$;

2. $f_1$ does not host $f_2$. In this case, the access request is redirected to the host of $f_1$ and all the way up until it reaches the host of $f_2$, which is then reduced to case 1. PipeLink proposes Hierarchical Linker Design (Section 4.4.2) for this case;
4.4.1 Hosted Function Calls

In Fig. 4.4, g hosts guest function f. Function g may directly call f, and may also indirectly call f via direct calls to other functions (in our example, g may call h which in turn may call f). We introduce a proxy hf for the indirect call of f through h. Any call to h is also treated as a potential call to f via the proxy hf. Multiple calls to h are treated as multiple calls to the proxy, each having a unique access port to f through the proxy.

Since g is the host of f, it has direct access to f’s I/O ports. PipeLink synthesizes the use-resource token network TN(g,f) for f as described in Section 4.3. Fig. 4.5 illustrates the construction, discussed in detail next.

Based on Fig. 4.1(c), the Resource block will be the guest function fi, and TN(h,fi) will be the desired Resource Sharing system.

Pipelined resource sharing requires deterministic control, and the ctrl tokens need to be generated in a way that the senders/receivers could interact with fi in the same order as specified in the software. These control tokens can be systematically generated using the induction on the structure of the software program.

Fig. 4.5(a) shows an example code for g from Fig. 4.4 which calls f three times: f0
(line 11), $f^1$ (line 12) and $f^2$ (line 16). Moreover, $h$ also accesses $f$ through its proxy function $hf$, which will redirect the access request to $g$ (host function of $h$). Therefore, $g$ also implicitly calls $f$ through the execution of $h$ (Line 14 and 19), marked as $f^3$ and $f^4$ respectively in Fig. 4.5(a).

From $g$’s point of view, it calls $f$ at five call sites, and the use-resource token network is constructed from the structure of the program: $(f^0, f^1)$ in sequential block $SEQ_1$, $f^2$ in loop block $LOOP_2$, $(f^3, LOOP_2)$ in sequential block $SEQ_2$ which is inside if block $IF_1$, and $(SEQ_1, IF_1, f^4)$ in sequential block $SEQ_0$. This is shown in Fig. 4.5(b).

Note that the construction from Section 4.3 will result in two unique access ports for $hf$ as illustrated in Fig. 4.5(b). Additional logic is needed to route tokens from the proxy $hf$ to the appropriate access ports for $f$. We discuss this construction in detail in the following section.

4.4.2 Nested Function Calls: incremental linking

If $h$ has a call to $f$ which it does not host, then $h$ does not have direct access to the I/O ports of $f$. In order to achieve the invocation, $h$ creates a proxy function $hf$ internally. This proxy simply forwards the access request to the host of $h$. If the host of $h$ also hosts $f$, then the access can be resolved there via the construction in Section 4.4.1. Otherwise, this forwarding process is repeated by $h$’s host until eventually $f$’s host is reached. We call this process incremental linking, because the final access circuit for $f$ is incrementally constructed by this forwarding process. The forwarding process is guaranteed to terminate at $f$’s host because of the following result:

**Theorem 5.** If $f_1$ (hosted by $m_1$) calls $f_2$ (hosted by $m_2$), and if $f_1$ does not host $f_2$, then $m_2 = m_1$, or $m_2$ dominates $m_1$ in the extended call graph.

**Proof.** We mark the entry node in the extended graph $ENTRY$. Suppose $m_2$ does not dominate or equal to $m_1$, then there exists a path for $ENTRY \rightarrow m_1$ without going through $m_2$,
noted as $path_1$. $m_1$ hosts $f_1$, so there exists a path for $m_1 \rightarrow f_1$. $f_1$ calls $f_2$, so there is a path for $f_1 \rightarrow f_2$. Consider the path $path_1 \rightarrow f_1 \rightarrow f_2$, it starts from $ENTRY$ and ends at $f_2$ without going through $m_2$, which contradicts with the assumption that $m_2$ hosts $f_2$. Therefore, $m_2$ dominates or equals to $m_1$. 

\[ \text{COL}(h,hf) \]
\[ hf_{in} \]
\[ \text{COL}(g,f) \]
\[ gf_{in} \]
\[ f \]
\[ f_{out} \]
\[ h \]
\[ \text{COL}(h,hf) \]
\[ inPort \]
\[ outPort \]

Figure 4.6: Incremental linking example

Fig. 4.4(a) shows code snippets for the nested invocations, which are handled by the incremental linker (i.e., the dash lines in Fig. 4.6). When the proxy $hf$ is used, unique access ports are created for each invocation of $hf$ even though $hf$ only has a single access port. Hence, PipeLink must construct the logic to route data from the proxy function $hf$ to each unique access port. We detail this design next, and illustrate each step of the construction in Fig. 4.7 and Fig. 4.8.

When $h$ tries to access $f$ (Fig. 4.6(a)), PipeLink creates a internal proxy function $hf$, which collects the input arguments to $f$ and sends them to $h$’s host function $g$. When $g$ receives the input arguments $hf_{in}$ from $h$, it treats them as normal input arguments to $f$, constructs the collection circuit $\text{COL}(g,f)$ as normal (Section 4.4.1) and directly connects the final input argument $gf_{in}$ to $f$.

When $h$ tries to access the memory bank $A$ (Fig. 4.6(b)), the circuit is constructed in a similar way. Although $g$ does not explicitly access $A$, it implicitly accesses $A$ through the invocation of $h$ (Line 8 and Line 13 in Fig. 4.4(a)). Therefore, when $g$ receives the input argument $hA_{in}$ from $h$, it also creates an internal proxy function $gA$ to handle the invocation to $A$. 

75
Fig. 4.7(a) shows the constructed use-resource token networks and datapaths for both functions $g$ and $h$, constructed according to Section 4.3. Note that the final access to $f$ is only partially resolved, as there is a proxy function $hf$ used for the call site of $f$ within $h$. $g$ directly/indirectly calls $f$ five times, of which two calls are via the proxy $hf$.

The collection circuit $COL(g,f)$ receives two input arguments $hf_{x0}$ and $hf_{x1}$ from $hf$, and the delivery circuit $DEL(g,f)$ sends two outputs $hf_{y0}$ and $hf_{y1}$ back to $hf$. Moreover, the token generators $T3$ and $T4$ in $TN(g,f)$ represent whether $h$ is currently accessing $f$ or not; those use-resource token streams must also be appropriately generated from $TN(h,f)$. PipeLink’s incremental linker constructs the datapath as well as use-resource token logic to complete the circuit.

**Datapath for incremental linking:**

Fig. 4.7(a) shows the generated circuits after PipeLink compiles $g$ and $h$ into separate hardware and generates $TN(g,f)$ and $TN(h,f)$ respectively. Since $h$ does not host $f$, it accesses $hf$ instead which will send the argument $hf_{in}$ (receive the result $hf_{out}$) to (from) $g$. To connect $hf_{in}$ to $hf_{x0}/hf_{x1}$, PipeLink introduces additional $DEL(g,h)$ logic to correctly route the data tokens to $f$, and $COL(g,h)$ to route the result back to $hf$ as illustrated in Fig. 4.7(b). We remark that because $g$ has two callers of $h$ (Line 8 and Line 13 in Fig. 4.4), the $DEL(g,h)$ has two target receivers: $hf_{x0}$ and $hf_{x1}$ respectively. A similar remark applies to $COL(g,h)$.

Furthermore, the token generators $T3$ and $T4$ in $TN(g,f)$ also come from $TN(h,f)$, and should be propagated from $h$ to $g$ through $DEL(g,h)$. Fig. 4.7(c) shows the linker datapath for propagating the control tokens. This completes the datapath for incremental linking.

**Control for incremental linking:**

The newly introduced datapath blocks from Section 4.4.2 require control inputs. The missing control tokens are illustrated in Fig. 4.8(a). Since the datapath blocks are $DEL(g,h)$ and $COL(g,h)$, it is natural to use $dp_{gh}$, the output generated by $TN(g,h)$ to control them. Unfortu-
Figure 4.7: Incremental linker datapath
Figure 4.8: Linker Design
nately, this is not correct, because this does not account for the possibility of repeated calls to \( f \) from \( h \).

Consider the example in Fig. 4.4 where \( h \) is called twice (Line 8 and Line 13) inside \( g \) (marked as \( h^0 \) and \( h^1 \) respectively). As a result, \( TN(g,h) \) will generate \( dp_{gh} \) twice, one for \( h^0 \) (marked as \( dp^0_{gh} \)) and one for \( h^1 \) (marked as \( dp^1_{gh} \)). During the invocation of \( h^0 \), suppose \( f \) is invoked twice inside \( h \), then \( hf_\times x(0) \) (from \( DEL_0(g,h) \)), \( hf_\times y(0) \) (for \( COL(g,h) \)) and \( t_{hf} \) (for \( DEL_1(g,h) \)) are all generated twice while there is only one control token \( dp^0_{gh} \); hence, this will lead to deadlock. A similar issue arises for the invocation of \( h^1 \) when \( f \) inside \( h \) is invoked either more than once or zero times during \( h^1 \).

Fig. 4.8(b) shows the Control Path design for the hierarchical linker. The \( CP(g,h,f) \) is the Control Path for the nested case when \( g \) calls \( h \) and \( h \) calls \( f \). It has two inputs: \( dp_{gh} \) and \( t_{hf} \), and generates two output control tokens: \( dp_{ghf} \) for the datapath token transfer (i.e., for \( DEL_0(g,h) \) and \( COL(g,h) \)) and \( t_{ghf} \) for the control token transfer (i.e., for \( DEL_1(g,h) \)).

\( CP(g,h,f) \) needs to track how many times \( h \) calls \( f \) for each invocation \( h_i \), and replicate the control token \( dp_{gh} \) accordingly. Suppose during invocation \( h_i \), \( h \) calls \( f \) for \( N_i \) times, then \( t_{hf} \) will be generated in the form of \( 1 \ldots 1 0 \), and \( CP(g,h,f) \) will leverage this to generate \( dp_{ghf} \) for \( N_i \) times and the token values all equal to \( dp_{gh} \). Furthermore, since \( t_{hf} \) is generated for \( N_i+1 \) times, \( CP(g,h,f) \) generate \( t_{ghf} \) for \( N_i+1 \) times whose values all equal to \( dp_{gh} \). Fig. 4.8(c) shows the complete hierarchical linker design for \( h \) calling \( f \) for the example in Fig. 4.6(a).

### 4.5 Applications

#### 4.5.1 Function Pointers

PipeLink handles regular function calls and memory access in a unified way, which permits it to translate programs that previously were unsupported by dataflow HLS translations.

A function pointer is used to route arguments/return values from/to the caller to/from different functions; similarly, a pipelined asynchronous memory \([25]\) uses address to route
address and data to the corresponding memory banks. Inspired by this, PipeLink handles data pointers (accessing memories) and function pointers (accessing functions) uniformly.

```c
// data pointer
A[M], B[N];
int *ptr;
if (c) {
    ptr = A;
} else {
    ptr = B;
}
y = ptr[i];

// function pointer
int (*fPtr)(int);
if (c) {
    fPtr = f;
} else {
    fPtr = g;
}
y = (*fPtr)(i);
```

Fig. 4.9 shows a data pointer example where `ptr` points to memory bank `A` (when `c` is true) and `B` (when `c` is false) conditionally. Similarly, it also shows a function pointer example where `fPtr` points to function `f` (when `c` is true) and `g` (when `c` is false) in the same way. PipeLink treats the memory banks and regular function blocks as equal, so these two programs have exactly the same behavior and PipeLink synthesizes the same control and data path for them.

PipeLink tracks the def-uses of pointers in the LLVM IR to compute a set of candidate memory targets for each data pointer, and a set of candidate functions for each function pointer. It also records the conditions associated with each binding. Each candidate memory/function is assigned an integer index, and this index is used to route data to/from the actual memory/function. Sharing of the memory/function is handled by the unified scheme outlined earlier.

### 4.5.2 Pointer Arguments

Some function calls could have pointer arguments, which is unsupported even by some commercial tools. PipeLink, however, could handle the function pointer arguments in a generic way. If a function `f` has a pointer argument `ptr`, then the `ptr` will ultimately point to one (or multiple) memory banks, which is unknown to `f`. To handle this, PipeLink treats the pointer argument as a fake memory bank (i.e. a fake function) `fake_{ptr}`, and construct
such nested function call chain: \( host_f \rightarrow f \rightarrow fake_ptr \). Similar to Section 4.5.1, PipeLink tracks the def-uses of pointers of LLVM IR to compute a set of candidate memory targets for each pointer, and construct the incremental linker accordingly.

1
2
3
4
5
6
7
8
9
10
11
12
13

\[
\text{void } f(\text{int } *p,...) \{ \\
... \\
... \text{ use } p ... \\
... \\
\}
\]

\[
\text{void } m() \{ \\
... \\
\text{int } A[M]; \\
... \text{ use } A[...]. . . \\
f(A); \\
... \\
\}
\]

\[\text{a) Pointer argument example} \quad \text{b) compilation + Token Network generation}\]

\[\text{c) linker design}\]

\[\text{Figure 4.10: Pointer argument}\]

Fig. 4.10 (a) shows the example code for function pointer arguments. \( f \) receives a pointer argument \( p \) and uses \( p \) internally. \( m \) creates an array \( A \), uses \( A \) and passes \( A \) to \( f \). \( f \) creates a fake function \( fp0 \) (\( p \) is the name of the pointer argument, 0 represents the sequence id of the pointer argument in the argument list), and PipeLink considers the access to pointer argument \( p \) as a nested function call \( m \rightarrow f \rightarrow fp0 \). Fig. 4.10 (b) shows the synthesized circuits after compilation and token network generation. \( f \) does not know where \( p \) actually points to, so it creates a proxy function \( fp0 \) to represent the physical
memory bank $A$. Fig. 4.10(c) shows the complete linker design for the nested function call $m \rightarrow f \rightarrow fp0$.

### 4.5.3 Separate Compilation

PipeLink divides the circuit synthesis into a compilation stage where each function is synthesized into separate hardware modules, and a linking stage where the incremental linker generates the “glue” logic to combine the modules into a whole program dataflow graph. The separation of compilation and linking enables PipeLink to incrementally compile a new function into a circuit and generate the linking circuit for it without re-synthesizing every function again.

![f_in \rightleftharpoons f \rightarrow f_out
h \rightleftharpoons hgf_in](a) Before

![g \rightleftharpoons ggf_in](b) After

**Figure 4.11: Incremental compilation example**

Suppose initially $h$ and $f$ are synthesized into circuits, and Fig. 4.11(a) shows the hardware modules and the hierarchical linker for them. Later on, $g$ is also synthesized. Instead of re-synthesizing the programs regarding $g$, $h$ and $f$, Fig. 4.11(b) shows that PipeLink only needs to synthesize function $g$ and the linker for ($g,h$) and ($h,f$).

PipeLink’s design enables an HLS approach where functions are pre-synthesized into a HLS library. The final program can be generated relatively quickly by re-using pre-synthesized logic combined with incremental linking.
4.6 Modular Dataflow Optimizations

PipeLink converts a program into a static dataflow graph using a set of standard building blocks: token source, token sink, \( n \)-way copy, two-input controlled merge (\texttt{MERGE}), two-input controlled split (\texttt{SPLIT}), \( n \)-input function (\texttt{FUNC}), and an initial token buffer. These components can be translated directly into either synchronous elastic circuits or asynchronous circuits; in this chapter we take the asynchronous circuit path. In either case, the complexity of the final implementation is directly related to the structure of the dataflow graph and the complexity of individual components. We perform several dataflow graph optimizations to improve the quality of the final implementation.

4.6.1 Basic Clustering

PipeLink is built on top of the LLVM compiler infrastructure, and hence operates on the LLVM IR. LLVM encodes expressions into three-address IR instructions, and the HLS engine maps each of them into individual \texttt{FUNC} operators resulting in a large number of such function units. Like many previous papers (e.g. [66, 79]), PipeLink clusters \texttt{FUNC} operators within a single basic block into a larger monolithic \texttt{FUNC} operator. This reduces the latching overhead and permits logic optimization across operators within a single basic block. The potential cost of this optimization is a throughput penalty, but our experience with standard benchmarks shows that this transformation does not introduce new bottlenecks.

A second clustering operation (also supported by previous tools) that we support is to identify opportunities for grouping a sequence of 2-way 	exttt{MERGE} blocks into an \( n \)-way \texttt{MERGE}. This transformation also requires computing a new control token for the \( n \)-way \texttt{MERGE}. We call this transformation 	exttt{MERGE flattening}, as it “flattens” a tree of 2-way \texttt{MERGE} blocks. A similar clustering operation is used for flattening 2-way \texttt{SPLIT} blocks. These two transformations are helpful for nested conditionals in a program, switch statements, and for optimizing the logic generated by the incremental linker.
4.6.2 Split/Merge Fusion

Consider a program that sets $y$ to $g(x)$ if a condition $c$ is true, and to $f(x)$ if the condition is false. If this is written using an if-statement, then the dataflow graph generated is shown in Fig. 4.12(a). This has the benefit that only one of $f$ or $g$ is evaluated, saving energy; however, if the two functions are very simple, then the extra overhead introduced by latches and dataflow control might outweigh the benefits of conditional execution. In the latter case, we can replace this sub-graph with a single $FUNC$ module that implements the C expression “$c ? g(x) : f(x)$”; we call this split/merge fusion.

To support this optimization, PipeLink identifies all tokens that are generated under the same control-flow condition. This can be done by dataflow graph analysis as follows: (a) for any dataflow $FUNC$ module, all inputs and outputs are generated under the same control-flow condition; (b) the control input token to a $SPLIT$ has the same control-flow condition as its data input; (c) the control input token to a $MERGE$ has the same control-flow condition as its data output. Finally, the control flow conditions of the two data outputs of a split correspond to the control flow condition of the data input combined augmented with either true or false based on the control token input value (similarly for the merge). Hence, a fusion opportunity is identified by the fact that the split and merge pair have the same control input, and their data outputs and inputs match.

4.6.3 Cross Basic-block Clustering

Once control flow conditions are identified as described above, PipeLink can also cluster $FUNC$ modules across basic block boundaries. For example, consider a program fragment of the form

\[c \rightarrow 0 \quad 1\]

\[f \rightarrow 0 \quad 1\]

\[g \rightarrow 0 \quad 1\]

\[y \rightarrow \]

Figure 4.12: Split and merge fusion opportunity
\[ x = a + b \times c; \]
\[ \textbf{if} \ (c > 0) \ { \ z = 1 } \]
\[ y = b \times c; \]

In this case, the assignments to variables \( x \) and \( y \) are generated under the same control-flow condition. PipeLink can cluster such \( FUNC \) elements together. While this preserves data-dependencies, there is one additional constraint that must be introduced for this particular cross-basic block clustering step. To understand why, consider the following modified example:

\[ x = a + b \times c; \]
\[ \textbf{if} \ (c > 0) \ { \ z = 1 + x } \]
\[ y = b \times c + z; \]

In this particular example, there is a potential clustering opportunity that groups the first and third assignments together. However, clustering also introduces a new synchronization point in the computation—in particular, a \( FUNC \) block receives all input tokens before producing all output tokens. This is acceptable for the earlier example, but not for the program above because there is a data-dependence between \( x \) (an output of the potential cluster) and \( z \) (an input to the potential cluster). This would result in deadlock. To avoid this problem, we check for paths through the dataflow graph from potential cluster outputs to cluster inputs, and partition the cluster to ensure that output to input dependencies are eliminated.

Finally, such output to input dependencies can also be introduced through function calls or memory access. To support modular optimization, a function specifies \( ordering \) constraints for its input and output ports: an ordering constraint between two ports means that there is a potential data-dependency from an input to an output. These ordering constraints are used to augment the cluster partitioning algorithm so that we can safely apply the cross basic-block clustering step without requiring whole program dataflow graph analysis.
4.7 Evaluation

We use Fluid in Chapter 3 to map each function in the full program into an independent circuit. Then, we use PipeLink (also implemented using the LLVM compiler framework) to generate the linking among circuits. We implement the resource-use token network (Section 4.4.1) and the incremental linker (Section 4.4.2) as LLVM passes [12]. These passes accept optimized LLVM IR as input and generate dataflow circuit graphs as their output.

We develop a standalone dataflow graph optimizer in C++, which takes in the generated dataflow graph from our HLS engine, applies dataflow optimizations (Section 4.6), and emits the optimized dataflow graph. Finally, we build on the asynchronous circuit back-end developed by [66] to map the optimized dataflow graphs into asynchronous circuits in a 28nm technology.

4.7.1 Setup

Circuit synthesis. Each dataflow graph component is translated into a unique pipeline stage, and the data transfers between pipelined stages use the bundled data protocol [70]. The control for each pipelined stage uses micro-pipelines [91]. The control circuits (one per dataflow operator type) are custom designed asynchronous logic, and the bundled-data datapath uses combinational logic for computation that is mapped to a standard cell library.

Simulation methodology. We built a discrete-event simulator to simulate the synthesized bundled data circuits with 4-phase handshake for process communication. To annotate the simulator, we use HSPICE to extract the control circuit performance metrics in a 28nm process technology. For the combinational logic, we used a commercial logic synthesis tool with a commercial 28nm standard cell library to determine delay/power/area. Synchronous results were obtained using the same cell library and logic synthesis tool.
Workloads. We use six applications (dfadd, dfmul, dfdiv, gsm, sha, adpcm) from an HLS benchmark suite [7]. In addition, we selected three widely used benchmarks: a differential equation solver (differential) from [76], and fft and fir from [35].

Comparison. We compare our tool against an academic HLS tool LegUp v4.0 [35] as well as two commercial HLS tools Commercial 1 and Commercial 2.

Metrics. We measure Delay (ps), Area (µm²), Energy (pJ) and Throughput (MHz). We do not show leakage power results, as they are always proportional to the area. We run each benchmark twenty times (the input data are provided by each benchmark, and there are dozens of them to cover different cases) and use averages across the runs to report benchmark statistics. We use the same methodology to collect results for both PipeLink and other HLS tools.

4.7.2 Experiment Results

Table 4.1: PipeLink Analysis

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># of elements (before clustering)</th>
<th># of elements (after clustering)</th>
<th>Token Network Area (%)</th>
<th>Merge/Split Area (%)</th>
<th>FUNC Area (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfadd</td>
<td>1260</td>
<td>533</td>
<td>2.40</td>
<td>39.36</td>
<td>35.56</td>
</tr>
<tr>
<td>dfmul</td>
<td>1308</td>
<td>798</td>
<td>3.61</td>
<td>39.34</td>
<td>42.59</td>
</tr>
<tr>
<td>dfdiv</td>
<td>1818</td>
<td>1148</td>
<td>3.48</td>
<td>41.92</td>
<td>40.99</td>
</tr>
<tr>
<td>gsm</td>
<td>4083</td>
<td>2370</td>
<td>2.85</td>
<td>22.38</td>
<td>61.60</td>
</tr>
<tr>
<td>sha</td>
<td>1713</td>
<td>879</td>
<td>2.94</td>
<td>21.33</td>
<td>17.58</td>
</tr>
<tr>
<td>adpcm</td>
<td>7189</td>
<td>3065</td>
<td>4.88</td>
<td>26.18</td>
<td>50.24</td>
</tr>
<tr>
<td>diff</td>
<td>41</td>
<td>23</td>
<td>0</td>
<td>5.80</td>
<td>85.83</td>
</tr>
<tr>
<td>fft</td>
<td>898</td>
<td>494</td>
<td>4.07</td>
<td>24.77</td>
<td>28.41</td>
</tr>
<tr>
<td>fir</td>
<td>203</td>
<td>109</td>
<td>2.20</td>
<td>19.79</td>
<td>49.40</td>
</tr>
<tr>
<td>geometric mean</td>
<td>1013</td>
<td>539</td>
<td>1.69</td>
<td>24</td>
<td>42</td>
</tr>
</tbody>
</table>

Comparison. Table 4.1 shows the detailed analysis of PipeLink. It synthesizes approximately 1000 dataflow elements (Fig. 3.2) for these benchmarks, and can reduce the number by half through operator clustering and Split/Merge fusion (Section 4.6). Table 4.2 shows the normalized performance numbers of PipeLink, Commercial 1 and Commercial 2 compared to Legup. Fig. 4.13 and Fig. 4.14 show the per-benchmark spider plots of normalized performance (relative to Legup) as well as the geometric mean of the normalized
Table 4.2: Normalized Performance (compared to Legup)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PipeLink</th>
<th>Commercial 1</th>
<th>Commercial 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Delay</td>
<td>Area</td>
<td>Energy</td>
</tr>
<tr>
<td>dfadd</td>
<td>0.37</td>
<td>1.78</td>
<td>0.02</td>
</tr>
<tr>
<td>dfmul</td>
<td>0.51</td>
<td>2.48</td>
<td>0.04</td>
</tr>
<tr>
<td>dfdiv</td>
<td>0.53</td>
<td>1.65</td>
<td>0.01</td>
</tr>
<tr>
<td>gsm</td>
<td>0.72</td>
<td>4.13</td>
<td>0.04</td>
</tr>
<tr>
<td>sha</td>
<td>0.61</td>
<td>1.44</td>
<td>0.10</td>
</tr>
<tr>
<td>adpcm</td>
<td>0.49</td>
<td>1.58</td>
<td>0.02</td>
</tr>
<tr>
<td>diff</td>
<td>0.91</td>
<td>0.58</td>
<td>0.08</td>
</tr>
<tr>
<td>fft</td>
<td>0.74</td>
<td>1.77</td>
<td>0.11</td>
</tr>
<tr>
<td>fir</td>
<td>0.88</td>
<td>1.03</td>
<td>0.15</td>
</tr>
<tr>
<td>geomean</td>
<td>0.62</td>
<td>1.61</td>
<td>0.05</td>
</tr>
</tbody>
</table>

performance across all HLS benchmarks. Note that we plot the inverse of the normalized throughput; hence, for all metrics, lower is better. These spider plots can better illustrate the trade-off among Delay, Area, Energy and Throughput.

PipeLink consistently has the best energy per benchmark. Compared with Legup, the minimum energy saving is $6.67X$ for fir, the maximum saving is $100X$ for dfdiv and the average saving is $20X$. Even compared with Commercial 1, PipeLink still saves energy by $12X$ on average. PipeLink synthesizes asynchronous circuits and only triggers the processes with input data tokens, which results in significant energy savings.

In most cases, PipeLink has the smallest delay ($0.62X$ on average) and the highest throughput ($1.64X$ on average). The improvement comes from two aspects: (i) PipeLink does extensive operator clustering, which results in lower latency circuits through logic optimization and specialization; (ii) when there are (nested-)if statements, the delay/throughput of asynchronous circuits depends on the activated processes at run-time while synchronous circuits are limited by the worse-case scenario. For differential benchmark, however, Commercial 1 ($0.61X$) is much faster than PipeLink ($0.91X$). This is because differential primarily consists of a loop. PipeLink has to synthesize multiple Merges and Splits for loop control [93, 66], and they increase the total delay and lower the throughput.

PipeLink increases the area by $1.61X$ on average. The penalty can be attributed to the synthesis of Merges and Splits for if and loop statements that remain unfused, and the collection and delivery circuits for accessing shared resources. Table [4.1] shows that
Figure 4.13: Normalized per-benchmark performance (I) in default setting: delay (D), area (A), energy (E), leakage power (L), and throughput-inv (1/T)
Figure 4.14: Normalized per-benchmark performance (II) in default setting: delay (D), area (A), energy (E), leakage power (L), and throughput-inv (1/T)
the area of merges and splits takes up almost 25% of the total circuit area on average. In addition, PipeLink does not currently include an operator scheduling phase, and hence datapath operators that might be shared by commercial HLS tools would be replicated in PipeLink.

**Inline vs non-inline:** By default, PipeLink does not inline functions (unless specified in the program) while other tools try to inline functions. In order to study the effect of inlining, we also ran the benchmarks with PipeLink-inline, Commercial2-noinline and Legup-noinline. We were unable to change the inlining behavior of Commercial 1. Fig. 4.15 shows the normalized performance compared to Legup (the blue dash line). Note that differential and fir only contain one function call, so inlining has minimal effect on these two benchmarks.

Inlining functions decreases the delay for all tools. For PipeLink, inlining has varying effects on area and energy (the numbers increase for some benchmarks and decrease for others). For PipeLink, inlining means more redundant operators are synthesized, but it also has more opportunities for clustering. Also, inlining eliminates the resource sharing logic.

Other tools, however, do not share resources at function (i.e., hardware module) bound-
aries, so disabling inlining results in the creation of many additional hardware modules. These modules are instantiated at multiple places by their callers. This results in a significant increase in both area and energy when inlining is disabled.

### 4.8 Conclusion

We propose PipeLink, a dataflow HLS engine to map programs into asynchronous dataflow circuits. It has a generic pipelined resource sharing mechanism that enables pipelined access to shared resources in static dataflow circuits. It leverages this mechanism to support memory access and (non-pipelined) function calls uniformly, and synthesize circuits in a modular way by separating compilation and linking steps. It supports function pointers, which could not be handled even by some of the commercial tools. PipeLink also applies a range of dataflow optimizations to improve the circuit performance. Results show that PipeLink can dramatically save energy, and improve delay and throughput performance.

In future work, we plan to focus on area reduction by exploring the sharing opportunities of $\textit{FUNC}$ blocks through operator scheduling. We also want to incorporate slack matching technique [20] in our design to further increase throughput.
Chapter 5

Future Work

5.1 Area Saving

The evaluation results for Fluid and PipeLink suggest that our dataflow HLS tool synthesizes larger circuits than Legup and the two commercial tools. The proposed dataflow optimization techniques already reduce the area by a large portion, yet more aggressive optimizations are needed.

Table 4.1 shows that the area of MERGEs and SPLITs take up almost a quarter of the total circuit area even after the SPLIT/MERGE flattening and fusion. To further reduce their area, we could synthesize biased MERGE and SPLIT for the loop statements. For example, in Fig. 3.4 (c), Fluid synthesizes a MERGE operator to conditionally select from the initial token $x$ (from outside the loop) and the loop-updated token $x_3$ (from within the loop body). Generally speaking, the loop would run for more than one iteration before it finishes. Therefore, the MERGE would select $x_3$ (i.e., the right data port) most of the time, resulting in a biased MERGE. Similarly, Fluid synthesizes a SPLIT to propagate $x_2$ to either the MERGE (when the loop continues) or the SINK (when the loop finishes), resulting in a biased SPLIT. The biased MERGE/SPLIT has a smaller area than the regular MERGE/SPLIT.

Furthermore, we want to explore the opportunities of operator sharing in dataflow circuits. Our system adopts operator clustering (Section 3.5 and Section 4.6) to try to
combine computations together and do logic optimizations for them, but it will synthesize unique pipelined processes even for the replicated computation blocks. Inspired by the operator sharing technique in static HLS, we want to identify the common computation patterns in the programs, synthesize ONE copy of the pipelined process for each of them and share it across multiple callers. Since the "shared operator" is a single process that keeps the order of input tokens, we could simply synthesize the MIXER network for non-deterministic sharing.

Lastly, we want to study the function inlining in detail. The inlining would duplicate the circuit elements, but it also gives us more opportunities to cluster operators across function boundaries and saves us from synthesizing the control/data path for function calls. The trade-off here needs to be thoroughly studied.

5.2 Slack Matching

Slack matching [26] is the technique of adding pipelined buffers to an asynchronous pipelined design in order to prevent stalls and improve performance.

Our system needs to do slack matching in multiple places:

- our tool could generate the MERGE/SPLIT pair under the same control token, and they could potentially block each other. For example, Fig. 4.3 shows the datapath for the shared function calls. It has MERGE tree to conditionally pass the right argument to the function, and SPLIT tree to conditionally pass the function result to the right receiver. The corresponding MERGE/SPLIT pair uses the same control token generated by the token network, and PipeLink needs to insert the FIFO buffers at the control ports of the SPLIT tree to prevent the stall. The size of the FIFO buffers depends on the pipelined depth of the function, and can be decided through slack matching technique;

- Fluid and PipeLink tries to cluster as many combinational operators into one merged
 FUNC block as possible, which gives us more opportunities for logic optimizations. In the end, the merged FUNC block would be mapped to one pipelined operator. Compared with the un-optimized operators, the new operator has lower pipelining and thus lower throughput. For example, in Fig. 3.18(b), Fluid-opt has lower throughput than Fluid for the arith benchmark (which has purely combinational computations). Slack matching technique could re-insert FIFO buffers into the merged operator if needed to increase its pipeline and throughput.
Appendices
Appendix A

Template Circuits for Token Networks

Section 4.3 uses four basic types of token networks for the base case, sequential composition, conditional composition and iterative composition for pipelined resource sharing. These four basic token networks are pre-designed template circuits, and we talk about their designs here.

**Token Generator.** For the base case where a shared function \( f \) has one call site, PipeLink would generate a Token Generator for this single invocation. The Token Generator outputs 1, 0, 1, 0... iteratively. Its CHP implementation is:

\[
1 - \text{ctrl} > [6, 1]\text{ctrl}
\]

\( \text{ctrl} \) is the generated use-resource sequence. The token generator has buffers for 6 tokens with an initial token 1 placed inside during reset.

**Sequential Token Network.** Suppose we have the sequential invocations to \( f \):

\[
\begin{align*}
1 & \quad y0 = f(x0); \\
2 & \quad y1 = f(x1);
\end{align*}
\]

The \( SEQ \) token network is synthesized as Fig.[A.1](a), which has the following behaviors:

- when T0 generates token 1, \( SEQ \) network would generate token 0 for the \( dpCtrl \), and
a) SEQ token network  

b) IF\_c token network

c) LOOP\_c token network

Figure A.1: Template circuits for token networks
token 1 for the cpCtrl;

- when T0 generates token 0, seq network would switch control to T1, and no dpCtrl or cpCtrl would be generated;

- when T1 generates token 1, seq network would generate token 1 for the dpCtrl, and token 1 for the cpCtrl;

- when T1 generates token 0, seq network would switch control to T0 and generate token 0 for the cpCtrl. Then, it would resume to the initial state;

**IF.\_c Token Network.** Suppose we have the conditional invocations to f:

```
if (c) {
  S
} else {
  T
}
```

The IF.\_c token network is synthesized as Fig. [A.1](b), which has the following behaviors:

- if c is true (code segment S is executed), then before S finishes accessing f, IF.\_c token network would generate token 1 for dpCtrl.\_if\_c and cpCtrl.\_if\_c for each access. At the end, it would generate token 0 for cpCtrl.\_if\_c;

- if c is false (code segment T is executed), then before T finishes accessing f, IF.\_c token network would generate token 0 (1) for dpCtrl.\_if\_c (cpCtrl.\_if\_c) for each access. At the end, it would generate token 0 for cpCtrl.\_if\_c;

**LOOP.\_c Token Network.** Suppose we have the iterative invocations to f:

```
while (c) {
  S
}
```
The $LOOP_c$ token network is synthesized as Fig. [A.1](c), which has the following behaviors:

- while $c$ is true, then before $S$ finishes accessing $\ell$, $LOOP_c$ token network would generate token 1 for $cpCtrl_{loop_c}$ for each access to $\ell$ from $S$. Then $c$ is false, $LOOP_c$ token network would generate a token 0 for $cpCtrl_{loop_c}$;
Bibliography


[4] https://ptolemy.berkeley.edu/papers/03/Cal/


[15] number of hardware engineers. [https://www.bls.gov/ooh/architecture-and-engineering/computer-hardware-engineers.htm 2]

[16] number of software engineers. [https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm 2]


[34] Steven M. Burns and Alain J. Martin. *Syntax-directed Translation of Concurrent Programs into Self-timed Circuits*, pages 35–50. 1988. [56]


[72] Razvan Nane, Vlad Mihai Sima, Bryan Olivier, Roel Meeuws, Yana Yankova, and
Koen Bertels. DWARV 2.0: A cosy-based c-to-vhdl hardware compiler. In Dirk
Koch, Satnam Singh, and Jim Tørresen, editors, 22nd International Conference on
Field Programmable Logic and Applications (FPL), Oslo, Norway, August 29-31,

[73] S. F. Nielsen et al. A behavioral synthesis frontend to the haste/tide design flow. In
ASYNC 2009. [19][56]

[74] Sune Fallgaard Nielsen et al. Towards behavioral synthesis of asynchronous circuits
[56]

[75] P. G. Paulin et al. Scheduling and binding algorithms for high-level synthesis. In

[76] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis
of asics. IEEE Transactions on Computer-Aided Design of Integrated Circuits and
Systems, 8(6):661–679, 1989. [46][87]


[79] Song Peng, David Fang, John Teifel, and Rajit Manohar. Automated synthesis for
asynchronous fpgas. In Herman Schmit and Steven J. E. Wilton, editors, Proceed-
ings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate
Arrays, FPGA 2005, Monterey, California, USA, February 20-22, 2005, pages 163–
173. ACM, 2005. [83]

[80] F. Peverelli et al. Oxigen: A tool for automatic acceleration of c functions into
dataflow fpga-based kernels. In IPDPSW, 2018. [12][14][20][22][56][57]

level synthesis of memory-intensive applications. In 23rd International Conference on
Field programmable Logic and Applications, FPL 2013, Porto, Portugal, September
2-4, 2013, pages 1–4. IEEE, 2013. [12][14][61]

[82] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Maris-
abel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubraman-
nian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjan
Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki
Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Ja-
yaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou,
Kyle A. Lucke, J. P. Maaninen, Ramon Macias, Maire Mahony, David Alexan-
der Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin
Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish
Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wach- 
sler, Andrew C. Walton, David A. Wickeraad, Alvin Wijaya, and Hon Kwan Wu. 
Warehouse-scale video acceleration: co-design and deployment in the wild. In Tim 
Sherwood, Emery Berger, and Christos Kozyrakis, editors, ASPLOS ’21: 26th ACM 
International Conference on Architectural Support for Programming Languages and 
Operating Systems, Virtual Event, USA, April 19-23, 2021, pages 600–615. ACM, 
2021. 1

Hardt, and Bernd Kleinjohann. The specification language specc within the PAR- 
ADISE design environment. In Bernd Kleinjohann, editor, Architecture and De- 
sign of Distributed Embedded Systems, IFIP WG10.3/WG10.4/WG10.5 Interna-
tional Workshop on Distributed and Parallel Embedded Systems (DIPES 2000), Oc-
tober 18-19, 2000, Schloß Eringerfeld, Germany, volume 189 of IFIP Conference 

[84] C. Rubattu et al. Dataflow-functional high-level synthesis for coarse-grained reconfig-
urable accelerators. IEEE Embedded Systems Letters. 20, 22, 56, 57

[85] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, 
(ISCA), pages 282–294, 2020. 1

[86] Jocelyn Sérot et al. CAPH: A Language for Implementing Stream-Processing Appli-
cations on FPGAs. 2013. 9, 12, 13, 20, 22, 56, 57

[87] Sharad Sinha and Thambipillai Srikanthan. High-level synthesis: Boosting designer 

[88] A.M. Sllame and V. Drabek. An efficient list-based scheduling algorithm for high-
Architectures, Methods and Tools, pages 316–323, 2002. 5


[90] Arun Subramaniyan, Jack Wadden, Kush Goliya, Nathan Ozog, Xiao Wu, Satish 
Narayanasamy, David Blaauw, and Reetuparna Das. Accelerated seeding for 
genome sequence alignment with enumerated radix trees. In 2021 ACM/IEEE 48th 
Annual International Symposium on Computer Architecture (ISCA), pages 388–401, 
2021. 1


19, 56


[101] Lingxi Wu, Rasool Sharifi, Marzieh Lenjani, Kevin Skadron, and Ashish Venkat. Sieve: Scalable in-situ dram-based accelerator designs for massively parallel k-mer


