Fluid: An Asynchronous High-level Synthesis Tool for Complex Program Structures

Rui Li, Lincoln Berkley, Yihang Yang, and Rajit Manohar
Computer Systems Lab, Yale University, New Haven, CT 06520, USA
rui.li@yale.edu, linc.berkley@yale.edu, yihang.yang@yale.edu, rajit.manohar@yale.edu

***Abstract*—Current high-level synthesis (HLS) tools that generate synchronous logic construct a state machine that schedules program operations in each clock cycle. Rather than this centralized approach, we are developing an HLS methodology tailored to high-performance asynchronous dataflow circuits building on prior work in dataflow synthesis. We propose a new solution to dataflow circuit generation needed when translating real-world programs with complex control flow. We implement our approach in the LLVM compiler framework, 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.**

***Index Terms*—asynchronous HLS, dataflow circuits**

I. Introduction

Technology scaling has brought us to the sub-10nm era and a regime of operation where single-thread performance on general-purpose microprocessors has stagnated. As a result, accelerating software programs using either field-programmable gate arrays (FPGAs) or by using custom accelerators has become an important area of investigation.

Using a traditional hardware description language to describe a computation is quite different from writing standard software, so converting a software program directly into a good FPGA/ASIC implementation is an arduous, time-consuming task. The goal of high-level synthesis (HLS) tools is to provide an automated method for translating conventional software into a hardware description language. Using HLS can significantly reduce the design time for accelerators that are derived from pre-existing software [23].

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 [15], [19], [20], [26], [35], [39], [40]. There has also been previous work in translating software programs into asynchronous circuits [9]. Furthermore, some synchronous HLS tools also synthesize latency-insensitive dataflow circuits [14], [21], [37]. Other tools use domain-specific languages and special pragmas to simplify the high-level synthesis problem [1], [4], [8], [13], [17], [24], [30]–[32]. 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 paper 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.

Fluid goes through a number of steps to translate C programs to asynchronous dataflow circuits. Starting from C, we use the LLVM compiler framework [2] to generate optimized LLVM IR (Intermediate Representation). Fluid analyzes the IR and modifies it to handle complex/irregular control structures. After this, a dataflow graph is generated and further optimized to improve the design, and then directly translated into asynchronous bundled-data circuits. We also compare Fluid against an academic HLS tool (LegUp [12]) and two different commercial HLS tools on a combination of micro-benchmarks and existing HLS benchmarks.

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 that 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 paper is organized as follows: Section II introduces the prior work that we build on. Section III presents how Fluid constructs dataflow graphs based on the control-flow graph (CFG), including support for irregular CFGs (Section III-D) Section IV describes the dataflow optimizations currently in Fluid. Section V evaluates Fluid against against three other HLS tools. We provide an overview of the large body of related work in Section VI.

II. Background

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 [36]. The CHP language was translated into a CFG, 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 [36]. STF combines concepts from both static single assignment (SSA) [16] form and static single information (SSI) [6] form into a unified analysis.

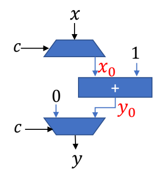
Consider the *if* example in Fig. 1a. In the true branch, x is conditionally used and y is redefined, so STF in Fig. 1b contains a *split* instruction which conditionally generates x_0 under condition c, and a *merge* instruction which conditionally selects y and y_0 under condition c. In the true branch, x_0 (not x) is used and the result is assigned to y_0 (not y).

```
int x;
int y=0;
if(c){
  y=x+1;
}
```

(a) *if code*

```
int x;
~~int y=0;~~
x_0 = split_c(x);
if(c){
  y_0 = x_0 + 1;
}
y_1 = merge_c(0, y_0);
```

(b) *STF form*



(c) *STF circuit*

Fig. 1: If Example.

[36] synthesizes two types of operators: *MERGE* and *SPLIT* for the *merge* and *split* instructions respectively. *MERGE* receives multiple inputs and sends one of them to the output port. *SPLIT* receives one input and sends it to one of the multiple output ports. The selection is controlled by the token received from the control port. For both operators, the left (right) port is selected under the false (true) condition. Fig. 1c shows the synthesized circuit for the *if* example. Apart from *SPLIT* and *MERGE*, a complete set of dataflow components needed to translate CHP programs includes a token source, token sink, copy, function computation component, and an initial token buffer [36]. STF does no optimizations to the synthesized dataflow circuits, and it only deals with simple control structures since its input is a CHP program.

Another commonly used operator is the uncontrolled *MERGE* (also called *MIXER*), which is similar to *MERGE*, except it does not have a control token port. It 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, the output is non-deterministic; hence, circuits using *MIXER*s often impose a mutual exclusion constraint on input token arrival so as to preserve deterministic execution. Compared with *MERGE*, the *MIXER* will decrease the circuit pipelining. In our work, we only use controlled *MERGE*.

To translate C programs, we leverage the production-quality LLVM open-source compiler framework [2]. 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 [2].

The standard data structure used for optimizing software programs is the control flow graph (CFG) [5]. 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.

III. Fluid Design

LLVM optimization passes read in an IR file, and modify the IR and CFG to improve the quality. Fluid is implemented as such pass. It reads in an optimized IR, applies new techniques discussed below to modify the CFG and IR so that the resulting program is equivalent to the original one and can be readily converted into static token form. Then it applies 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 [36]. The challenging part of STF generation is creating the *SPLIT* and *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 [5], 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 *delivery circuit* to create a conditional copy of the variable; symmetrically, it constructs a *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 uses 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.

We impose a *canonical form* requirement on CFGs which consists of two parts: (i) every loop has a single-entry and single-exit point; and (ii) if a basic block has multiple predecessors, then it must post-dominate its immediate dominator. Section III-D 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 [36], and ignore those edges in the CFG for condition extraction below.

A. Condition Extraction

Fluid first extracts the conditions between different basic blocks in a CFG. 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 an if/loop-block), it has one enter (exit) block that dominates (post-dominates) all the internal blocks.

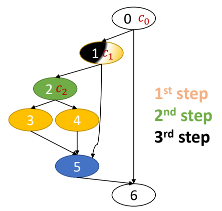


Fig. 2: CFG for condition extraction.

Fig. 2 shows a CFG that we will use as a running 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. The CFG has three branching variables c_0, c_1 and c_2 in B_0, B_1 and B_2 respectively.[1]

Given a block BB, Fluid uses reverse breadth-first-search along its predecessors to explore all the paths into BB until it encounters its immediate dominator. Consider B_5 as a starting point. In the first step, 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, Fluid further explores $\{B_3, B_4\}$'s predecessor B_2. B_1 is the immediate dominator of B_5, so the search stops at it. Now Fluid records the conditions for $B_2 \rightarrow B_5$: $\{c_2{=}0\}, \{c_2{=}1\}$. In the third step, 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 $\{c_1{=}0, c_2{=}1\}$.

Since loops in a canonical CFG are single-entry/single-exit, conditions involving loops are only considered internally for continuing/exiting the loops as well as the first iteration of the loop [36], and don't affect outside basic blocks.

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 differ in one condition which is complementary in the two chains[2]. Fig. 2 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 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 $src \rightarrow loop1_{exit} \rightarrow loop2_{entry} \rightarrow dst$, and repeat the argument for each segment. The case when only one of src/dst belong to a loop is similar. □

Lemma 2. *If dst unconditionally connects [3] 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 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 $src \rightarrow dst'$ into two parts: $src \rightarrow iDom$ and $iDom \rightarrow dst'$. There is one merged path for $iDom \rightarrow dst'$, so we truncate $src \rightarrow dst'$ to $src \rightarrow iDom$. By repeating this, we eventually reduce Case 2 to Case 1. □

B. Delivery and Collection Circuit Construction

For a token defined in src block and used in dst block, Fluid construct a delivery circuit to conditionally propagate it.

Calculating the delivery conditions. In Section III-A, Fluid records the entering conditions into dst from its predecessors that are dominated by its immediate dominator $iDom$. If $src \rightarrow dst$ can be found, we can directly get the delivery conditions. Otherwise, we divide $src \rightarrow dst$ into $src \rightarrow iDom$ and $iDom \rightarrow dst$. The conditions for $iDom \rightarrow dst$ is known, so we just need to calculate the conditions for $src \rightarrow iDom$ by applying the same rule iteratively. Based on Theorem 3, there exists only one condition chain for any $src \rightarrow dst$, so we can simply append these conditions together to form the final conditions.

Synthesizing the delivery circuit. Fluid synthesizes $SPLIT$s for each unique condition variable in the delivery conditions and connects them following the same order.

Fig. 3a shows the same CFG as in Fig. 2. 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. 3b, Fluid synthesizes $SPLIT_0$ (in B_0) to generates x_1 for B_1 when $\{c_0{=}0\}$, and $SPLIT_1$ (in B_1) to generate x_2 for B_2 when $\{c_1{=}0\}$.

Collection Circuit Construction.

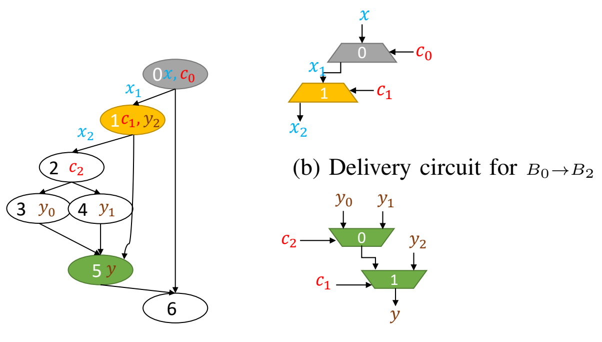
If token y in dst has multiple reaching definitions $y_1, y_2, ..., y_n$ in dst's n predecessors, Fluid synthesizes the

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

[2]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.

[3]If there is a path between src and dst, then we say *src connects to dst*.

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 $MERGE$s for each unique condition variable in the collection conditions and connects them in the reverse order of the collection conditions.



(a) *CDFG* (b) Delivery circuit for $B_0 \to B_2$ (c) *Collection circuit for* B_5

Fig. 3: Delivery and collection circuits.

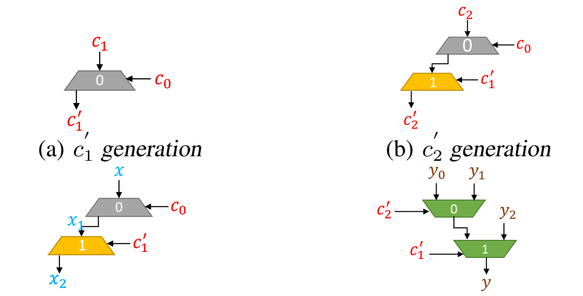
In Fig. 3a, B_5 receives $\{y_0, y_1, y_2\}$ from $\{B_3, B_4, B_1\}$, and assigns the final value to y. The collection conditions are:

$$B3 \to B5 : \{c_1 = 0, c_2 = 0\} \quad B4 \to B5 : \{c_1 = 0, c_2 = 1\} \quad B1 \to B5 : \{c_1 = 1\}$$

Fig. 3c shows the synthesized $MERGE$ tree in B_5.

C. Control Token Generation

The delivery and collection circuits consist of $MERGE$s and $SPLIT$s that require the control tokens, which could also need delivery/collection circuits if used/defined conditionally.



(a) c_1' *generation* (b) c_2' *generation* (c) *New delivery circuit* (d) *New collection circuit*

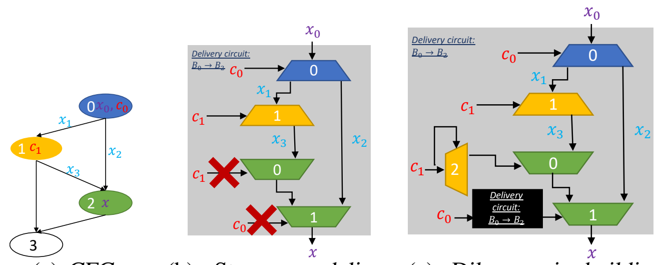
Fig. 4: Control token generation.

In Fig. 3a, suppose c_0, c_1 and c_2 are all defined in B_0. Then Fluid will conditionally generate c_1' for $B_0 \to B_1$ (Fig. 4a) and c_2' for $B_0 \to B_2$ (Fig. 4b) as well as the new delivery circuit for $B_0 \to B_2$ (Fig. 4c) and the collection circuit for B_5 (Fig. 4d).

D. Handling Non-canonical CFGs

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

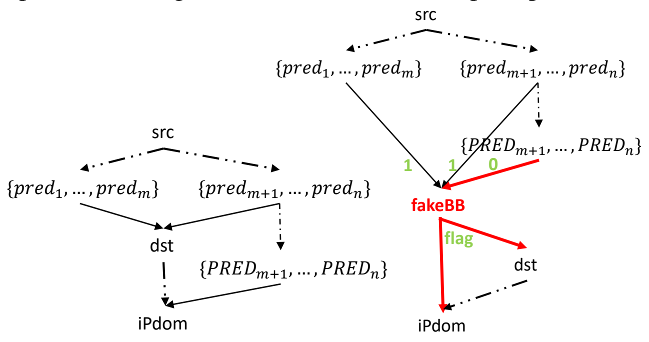
Fig. 5a shows a CFG with four basic blocks, and c_0 and c_1 are the condition variables for B_0 and B_1 respectively. x_0 is defined in B_0 and used in B_2, so and needs a delivery circuit. Fig. 5b shows the delivery circuit for $B_0 \to B_2$. We would create $SPLIT_0$ (in B_0) to conditionally generate 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 B_2) cannot directly use them. In Fig. 5c, we attempt to introduce $split_2$ to


Delivery circuit: $B_0 \to B_2$
Delivery circuit: $B_0 \to B_2$

(a) *CFG.* (b) *Strawman delivery circuit.* (c) *Dilemma in building the delivery circuit.*

Fig. 5: *Multi-Path Example.*

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 $B_0 \to B_2$—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. Fig. 6a illustrates the multi-path problem in a


src
$\{pred_1, ..., pred_m\}$
$\{pred_{m+1}, ..., pred_n\}$
dst
$\{PRED_{m+1}, ..., PRED_n\}$
iPdom
fakeBB
flag

(a) *Illustration.* (b) *Solution.*

Fig. 6: *Multi-Path Problem and Solution.*

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 $src \to dst$ which cannot be merged into one path. By Theorem 1, dst cannot post-dominate src, so it cannot post-dominate its n direct predecessors between $src \to dst$ either.[4] 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 exist paths [5] 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. 6b 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 $\{pred_1, ..., ped_n\}$ 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 $src \to fakeBB$.

[4]Note that dst could have other predecessors that do not belong to the paths for $src \to dst$, and they are not relevant to the multi-path problem.

[5]The paths between a set and a node dst are the collection of paths between each node in the set and dst.

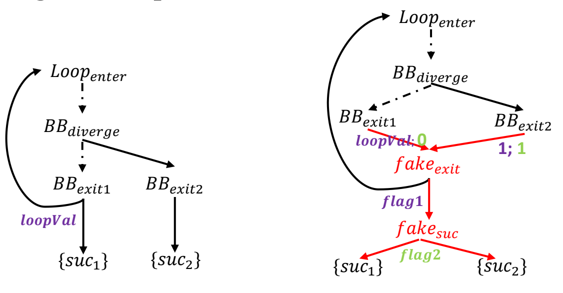
To preserve the correctness, we add edges from $fakeBB$ to $iPdom$ and dst respectively, and a fresh condition variable $flag$ which takes inputs from its direct predecessors $\{pred_1,...,pred_n,PRED_{m+1},...,PRED_n\}$. If $flag$ is true, $fakeBB$ jumps to dst; if $flag$ is false, $fakeBB$ jumps to $iPdom$.

$\{pred_1,...,pred_n\}$ 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 $iPdom$. The modified CFG has the same behavior as the original one.

The above transformation reduces the number of (src,dst) pairs that cause the multi-path problem in a CFG. As Fig. 6b shows, we added a new block $fakeBB$, and three groups of new connections: $\{PRED_{m+1},...,PRED_n\}{\rightarrow}fakeBB$, $fakeBB{\rightarrow}iPdom$ 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 new multi-path pair. Therefore, our solution can eliminate one multi-path pair ($src{\rightarrow}dst$ in Fig. 6a).

Note that the $fakeBB$ does not add any additional computation instructions; instead, it just serves as a intermediate step when transferring tokens for $src{\rightarrow}dst$, and the overhead is minimum.

Irregular Loops. We handle the case where a loop has more than one exit block. Fig. 7a shows the irregular loop. When the loop condition variable $loopVal$ is 1, the loop exit block BB_{exit1} exits the loop and jumps to its successor set $\{suc_1\}$; otherwise the loop continues. However, starting from $BB_{diverge}$, there is a second exit block BB_{exit2}. Fig. 7b shows the solution. We create two new blocks $fake_{exit}$ and $fake_{suc}$. The new loop condition variable is $flag1$. If BB_{exit2} is executed, $flag1$ becomes 1 and the loop exits; otherwise $flag1$ equals to $loopVal$. Therefore, the new CFG has the same behavior of running/exiting the loop as the original. When the loop exits, $fake_{exit}$ jumps to $fake_{suc}$. If the loop exits from BB_{exit1}, $flag_2$ equals to 0 and $\{suc_1\}$ will be executed. If the loop exits from BB_{exit2}, $flag_2$ equals to 1 and $\{suc_2\}$ will be executed. Therefore, the new CFG has the same behavior after exiting the loop as the original one.



(a) *Illustration.* (b) *Solution.*

Fig. 7: *Irregular loop and Solution.*

Note that Fluid cannot handle the loops with more than one entry blocks, which could be generated from **goto** statements.

A canonical CFG—an assumption implicit in previous work like [36]—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.

IV. 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.

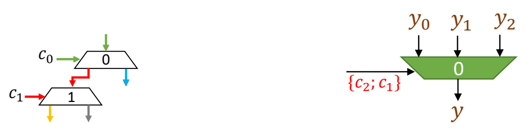
A. Operator clustering

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.

The control nodes (i.e., MERGE and SPLIT) will divide the whole graph into distinct control regions, and operator clustering is only applicable to nodes within the same control region. To identify them, we assign colors to the graph edge based on the condition it is activated. The in/out edges to a function node have the same color, but the MERGE and SPLIT node will update the output edge color from the input edge color. The function nodes whose output edges have the same color can now be safely clustered.

B. MERGE and SPLIT Tree Flattening

In Section III-B, Fluid synthesizes MERGEs and SPLITs for each unique condition variable in the collection and delivery conditions, potentially generating a tree of 2-way MERGEs and SPLITs. Fluid further flattens them into the N-way MERGE and SPLIT, which reduce the delay, area and energy consumption. The control for the flattened block is generated by tracing the original control tokens to regions that have the same color as the single data input (SPLIT) or output (MERGE) and computing the appropriate multi-bit control value.



(a) *Edge coloring* (b) *3-way* MERGE

Fig. 8: *Dataflow optimizations.*

Fig. 8a shows the edge coloring for the delivery circuit (Fig. 3b). Fig. 8b shows the 3-way MERGE synthesized for the collection circuit (Fig. 3c).

V. Evaluation.

A. Control Circuit Synthesis

Each dataflow graph component is translated into a unique pipeline stage, and the data transfers between pipelined stages

TABLE I: Performance of LegUp 4.0 and two commercial HLS tools

	Commercial 1					*Commercial 2*					Legup				
Benchmark	**Delay**	**Area**	**Energy**	**Leak.**	**Throu.**	**Delay**	**Area**	**Energy**	**Leak.**	**Throu.**	**Delay**	**Area**	**Energy**	**Leak.**	**Throu.**
adpcm-u	6000	9846	81	4616	167	3750	6825	22	2123	267	6000	8013	66	3405	167
dfadd-a	6000	8780	55	3282	167	6250	12154	54	3241	160	9000	10941	77	2964	111
differential	130000	8771	1861	4208	8	217778	25491	6732	8907	5	203333	15948	3451	5061	5
gsm-d	20667	2349	55	776	48	23333	9502	175	2813	43	28750	9682	176	2934	35
mpeng-d	6000	4096	28	1651	167	45000	11305	470	3587	22	42222	6327	267	2565	24
arith	5500	8584	83	4512	182	6250	13026	88	4106	160	11250	13156	204	5074	89
if	4000	3128	25	1273	250	3750	10482	41	3006	267	10000	15342	153	4342	100
for0	10500	2082	22	800	95	6250	4396	25	1376	160	16667	3974	47	1237	60
for1	25600	3447	149	1421	39	6250	3949	23	1245	160	35000	4065	127	1220	29
if-loop	11000	4699	86	1960	91	7500	14034	105	4119	133	13750	18772	257	5432	73
Ratio1	0.53	0.61	0.43	0.73	1.87	0.83	1.22	0.95	1.10	1.20	1	1	1	1	1
Ratio2	0.59	0.43	0.42	0.59	1.69	0.38	0.89	0.34	0.86	2.65	1	1	1	1	1

use the bundled data protocol [25]. The control for each pipelined stage uses micro-pipelines [34].

Fig. 9 shows a standard bundled data circuit template that we use in our evaluations [33]. 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 circuity implements the four-phase handshake using a Muller C-element (C). When the *input* token is ready ($in.rdy$ signal is high), and the successor stage is empty ($out.ack$ 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 $out.rdy$ signal is set to high. After the next stage captures this data, it will set the acknowledge signal $out.ack$ to high, allowing the current stage to reset. Delay lines (15% slower than the worse-case delay of the stage logic) are added to ensure successful data capturing and processing.

B. 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. 9. Each pipelined process fetches data from the predecessors and sends out result 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 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. 9). Synchronous results were obtained using the same cell library and same commercial logic synthesis tool.

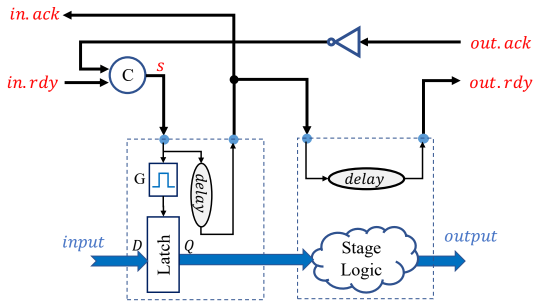


Fig. 9: Bundled-data circuit template.

C. Experimental setup and results

We synthesize the following microbenchmarks: (i) *arith*, which calculates $y = (x_0 + x_1) \times (x_2 + x_3) + (x_4 \times x_5) \times (x_6 \times x_7)$; (ii) *if*, with the true branch does addition and the false branch does division. The true branch will be triggered; (iii) *for0 and for1*. *for0* has a single loop and *for1* has two nested loops. Both count the number of iterations; (iv) *if-loop* which has an if statement: the true branch has a one-layer loop (does counting) and the false branch does division. The true branch is triggered.

We also extract five kernel functions from five applications which are mostly taken from an HLS benchmark suite [3] or used in synchronous ASIC synthesis benchmarking [28]: (i) *differential*, a differential equation solver [28]; (ii) *adpcm-u*, the uppol2 function from adpcm [3]; (iii) *dfadd-a*. The add function from dfadd [3]; (iv) *gsm-d*. The gsm_div function from gsm [3]; (v) *mpeg-d*. The decode function from mpeg [3].

We compare our tool with *LegUp* v4.0 which is a commonly used academic HLS tool, and two commercial HLS tools *Commercial 1* and *Commercial 2*. Furthermore, we have *Fluid* (vanilla version of Fluid) and *Fluid-opt* (Fluid with dataflow optimizations). We use the following performance metrics: Delay (ps), Area (μm^2), Energy (pJ), LeakPower(nW) and Throughput (MHz). We run each benchmark twenty times with random data 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.

Table II shows the performance of our system, and Table I shows other tools. Each table has three sections: the first shows the performance of HLS benchmarks, and the second shows microbenchmarks. To summarize across benchmarks, we use the geometric mean of normalized performance compared to *LegUp*; Ratio1 corresponds to the HLS benchmarks, and Ratio2 corresponds to microbenchmarks.

Delay. Fluid has longer delay for two reasons: 1) it fails to do logic optimizations for operator clusters; 2) it synthesizes $MERGE$s and $SPLIT$s which contribute to the extra delay. Fluid-opt can avoid the extra delay from reason 1. 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). Fluid-opt reduces delay by 1.64X and 1.92X for HLS benchmarks and microbenchmarks respectively.

Area and Leakage. Fluid has overhead for control and $MERGE$s and $SPLIT$s, but it avoids constructing a global state

TABLE II: Fluid and Fluid-opt Performance

	Fluid					Fluid-opt				
Benchmark	**Delay**	**Area**	**Energy**	**Leak.**	**Throu.**	**Delay**	**Area**	**Energy**	**Leak.**	**Throu.**
adpcm-u	11104	33814	84	10331	367	4588	7381	10	2424	560
dfadd-a	9445	132021	31	35081	212	7415	64258	8	18905	278
differential	226516	19051	1545	6217	4	165814	11180	1007	3365	6
gsm-d	41378	6301	95	1721	25	26861	5274	51	1607	39
mpeng-d	15844	22233	24	7147	99	7523	7285	5	2231	193
arith	4378	10872	20	3634	563	3548	9268	29	3058	289
if	1858	8506	2	2926	854	1566	1650	1	639	1754
for0	19680	2596	55	755	52	13628	2080	18	714	80
for1	40079	4927	175	1410	26	30843	4356	61	1341	36
if-loop	19946	10428	56	3469	52	14458	3410	18	1252	81
Ratio1	1.03	2.68	0.41	2.35	1.63	0.61	1.19	0.12	1.08	2.50
Ratio2	0.68	0.73	0.21	0.74	1.99	0.52	0.38	0.11	0.42	2.54

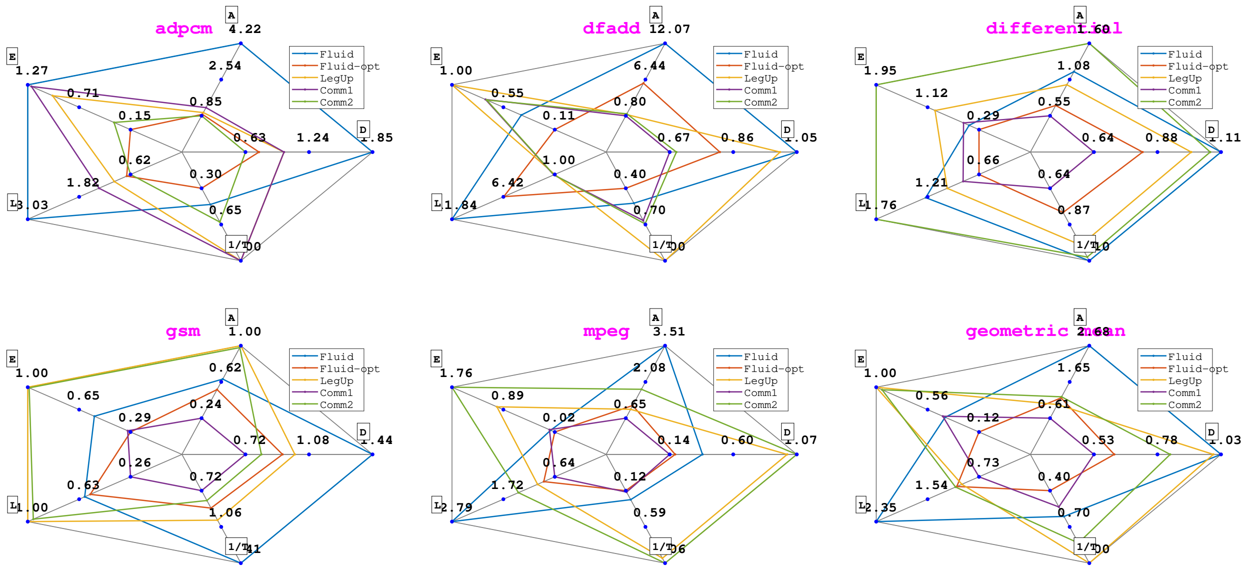


Fig. 10: *Normalized per-benchmark performance: delay (D), area (A), energy (E), leakage power (L), and throughput-inv (1/T)*

machine to control program execution. Fluid-opt significantly improves area compared to Fluid due to operator clustering. Fluid-opt increases HLS benchmark area by 1.19X, while reducing it by 2.63X for microbenchmarks. The area penalty is particularly severe for *dfadd-a*, which has many nested if and loop statements, increasing control overhead. Leakage power results are qualitatively similar to those for area.

Energy. Our tool pays extra energy for control circuits, but it saves energy by only triggering the processes that receive the input data. Fluid-opt reduces energy by 8.33X and 9.09X for HLS benchmarks and microbenchmarks compared to LegUp.

Throughput. Our tool synthesizes pipelined circuits, increasing throughput. Note that operator clustering could increase throughput by reducing loop latency, but also reduces pipelining which harms throughput. The result shows that Fluid-opt has higher throughput than Fluid for most benchmarks. Fluid-opt increases the throughput by 2.5X for HLS benchmarks, and 2.54X for microbenchmarks compared to LegUp.

Overall performance Fig. 10 shows per-benchmark spider plots of normalized performance as well as the geometric mean of the normalized performance of HLS 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.

VI. Related work

Asynchronous synthesis. [7], [11], [18] are based on syntax-directed translation of the syntax of a message-passing hardware description language (HDL) into an asynchronous circuit. [40] uses Petri-net based synthesis of timed circuits from a message-passing HDL. [26], [27] 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.

[19] proposes a source-to-source transformation with concurrency optimizations. [20] proposes a new scheduling algorithm for generating asynchronous design out of synchronous one by removing the discrete time assumption. [36], [39] synthesizes circuits in a data-driven manner using CFGs, targetting pipelined processes. [35] 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.

[9], [38] compiles C programs to Pegasus [10] IR, which is later synthesized to 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 III.

Dataflow HLS. Some synchronous HLS tools generate elastic dataflow circuits. [14], [21], [22], [37] directly map software programs to dataflow circuits. Fluid synthesizes controlled MERGE (instead of MIXER) to control circuit execution, which enables better circuit pipelining. Besides, Fluid can handle more complex program control structures when using controlled MERGE. Some other works propose domain-specific languages [1], [4], [8], [13], [31], [32] and special directives/pragmas [17], [24], [30] to help the synthesis. Fluid, however, does not require any changes to the source code or language-level support.

[14] propose a protocol for implementing elastic communication channels for synchronous systems. [37] uses syntax-directed translation to map functional programs into dataflow circuits leveraging the functional programming model. [21] generates elastic circuits, and optimizes for memory access. CAPH [32] is a domain-specific language for streaming applications. [31] extends [32] for CGRA. CAL [1] is designed for dataflow programming with actors, and OpenDF [8] converts CAL code to HDL. [13] generates elastic circuits for stencil computation. Fluid uses standard C as input, includes optimizations for SPLITs/MERGEs, and handles complex CFGs.

[17] uses directives to control array partitioning, inlining options, etc. for mapreduce programs. [24] generates HLS directives to partition the program into different clock domains. [30] uses LLVM to generate IR from C programs, applies optimizations such as vectorization and loop unrolling based on resource models. After that, these works [17], [24], [30] 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 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).

VII. Summary

We propose a new solution to dataflow circuit generation that can handle complex control structures with several dataflow optimizations. We compare against other HLS tools and show improvements in terms of energy and throughput. In future work, we plan to incorporate additional dataflow graph optimizations to further improve the quality of results. Also, we plan to explore synchronous implementations (e.g. [29]) to evaluate synchronous/asynchronous trade-offs.

Acknowledgment

The research in this paper was supported in part by DARPA IDEA grant FA8650-18-2-7850, and in part by DARPA POSH grant HR001117S0054-FP-042.

References

[1] https://ptolemy.berkeley.edu/papers/03/Cal/.
[2] https://llvm.org/.
[3] http://www.ertl.jp/chstone/.
[4] M. Abid et al. System level synthesis of dataflow programs: Hevc decoder case study. In *Proceedings of the 2013 Electronic System Level Synthesis Conference*, 2013.
[5] A. V. Aho et al. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition.
[6] C. S. Ananian and M. Rinard. Static single information form. Technical report, M.S. thesis, MIT, 1999.
[7] Berkel et al. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*. Cambridge International Series on Parallel Computation. Cambridge University Press, 1994.
[8] S. S. Bhattacharyya et al. Opendf: a dataflow toolset for reconfigurable hardware and multicore systems. *SIGARCH Comput. Archit. News*.
[9] M. Budiu et al. Spatial computation. In *ASPLOS 2004*.
[10] M. Budiu et al. Pegasus: An efficient intermediate representation. Technical report, Carnegie Mellon University, 2002.
[11] S. M. Burns and A. J. Martin. *Syntax-directed Translation of Concurrent Programs into Self-timed Circuits*, pages 35–50. 1988.
[12] A. Canis et al. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *FPGA 2011*.
[13] Y. Chi et al. Soda: Stencil with optimized dataflow architecture. In *ICCAD*, pages 1–8, 2018.
[14] J. Cortadella et al. Synthesis of synchronous elastic architectures. In *DAC 2006*.
[15] J. Cortadella et al. Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2006.
[16] R. Cytron et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 1991.
[17] D. Diamantopoulos et al. High-level synthesizable dataflow mapreduce accelerator for fpga-coupled data centers. In *SAMOS*, 2015.
[18] D. A. Edwards et al. Balsa: An asynchronous hardware synthesis language. *Comput. J.*, 45(1):12–18, 2002.
[19] J. Hansen et al. Concurrency-enhancing transformations for asynchronous behavioral specifications: A data-driven approach. In *ASYNC 2008*.
[20] J. Hansen et al. A fast branch-and-bound approach to high-level synthesis of asynchronous systems. In *ASYNC 2010*.
[21] L. Josipovic et al. Dynamically scheduled high-level synthesis. In *FPGA 2018*.
[22] L. Josipovic, A. Guerrieri, and P. Ienne. Synthesizing general-purpose code into dynamically scheduled circuits. *IEEE Circuits and Systems Magazine*, 21(2):97–118, 2021.
[23] D. Koch et al., editors. *FPGAs for Software Programmers*. Springer.
[24] T. Liang et al. Hi-clockflow: Multi-clock dataflow automation and throughput optimization in high-level synthesis. In *ICCAD*, 2019.
[25] C. Mead et al. *Introduction to VLSI systems*. Addison-Wesley, 1980.
[26] S. F. Nielsen et al. A behavioral synthesis frontend to the haste/tide design flow. In *ASYNC 2009*.
[27] S. F. Nielsen et al. Towards behavioral synthesis of asynchronous circuits - an implementation template targeting syntax directed compilation. In *DSD 2004*.
[28] P. G. Paulin et al. 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.
[29] A. Peeters et al. Synchronous handshake circuits. In *ASYNC 2001*.
[30] F. Peverelli et al. Oxigen: A tool for automatic acceleration of c functions into dataflow fpga-based kernels. In *IPDPSW*, 2018.
[31] C. Rubattu et al. Dataflow-functional high-level synthesis for coarse-grained reconfigurable accelerators. *IEEE Embedded Systems Letters*.
[32] J. Sérot et al. *CAPH: A Language for Implementing Stream-Processing Applications on FPGAs*. 2013.
[33] J. Sparso et al. *Principles asynchronous circuit design*. Springer.
[34] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 1989.
[35] S. Taylor et al. Automatic compilation of data-driven circuits. In *ASYNC 2008*.
[36] J. Teifel et al. Static tokens: using dataflow to automate concurrent pipeline synthesis. In *ASYNC 2004*.
[37] R. Townsend et al. From functional programs to pipelined dataflow circuits. In *Proceedings of the 26th International Conference on Compiler Construction, 2017*.
[38] G. Venkataramani et al. C to asynchronous dataflow circuits: An end-to-end toolflow. In *IWLS*, 2004.
[39] C. G. Wong and A. J. Martin. High-level synthesis of asynchronous systems by data-driven decomposition. In *DAC 2003*.
[40] T. Yoneda et al. High level synthesis of timed asynchronous circuits. In *ASYNC 2005*.