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

Rui Li
Intel
rui.lee92@gmail.com

Lincoln Berkley
Yale University
linc.berkley@yale.edu

Rajit Manohar
Yale University
rajit.manohar@yale.edu

***Abstract*—Dynamically scheduled high-level synthesis (HLS) is an approach to HLS that maps programs into dataflow circuits. These circuits use distributed control for communication and therefore can be automatically pipelined. However, pipelined function call is challenging due to the absence of centralized control, thus general software programs cannot be well-supported by the HLS tools. Traditional solutions to this problem impose restrictions on pipelining when accessing the shared functions. We present PipeLink, a modular synthesis method that decomposes the synthesis into compilation stage and linking stage, which enables the fully-pipelined access to shared functions in dataflow circuits. We develop a complete HLS engine using this approach and use asynchronous circuits as the target implementation. Our engine supports pipelined access to shared function units, as well as pipelined memory access in a unified fashion. Compared to existing HLS tools, PipeLink results in 20X reduction in energy and 1.56X improvement in throughput.**

I. Introduction

High-level synthesis (HLS) is a design methodology that automatically maps software programs (e.g., C/C++) into a hardware description language that implements the same functionality. It has received a lot of interests in hardware accelerator design [26], [34] since it can reduce design time and is more accessible to software developers. There are two major categories of HLS: statically scheduled HLS (static HLS) and dynamically scheduled HLS. Statically scheduled HLS [17], [27] converts a program into a datapath that contains all the operators from the program and a global state machine that enforces centralized control for these operators. To synthesize the state machine, the static HLS tool needs accurate delay information (in terms of clock cycles) for all operators at synthesis time and uses this information to compute an optimized schedule. This early scheduling may lead to sub-optimal performance, but the central scheduler design makes it good at coordinating different circuit components and managing resource access conflicts.

Dynamically scheduled HLS [14], [23], [24], [28], [33], [38], on the other hand, maps programs into dataflow circuits consisting of concurrent, independent dataflow elements. The dataflow elements communicate with each other through local control (i.e., handshake or ready/valid protocols), so they can be automatically pipelined without requiring global control. However, the lack of global control makes it hard to coordinate the actions of different dataflow elements.

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.

In this work, we argue that currently both static HLS and dynamically scheduled HLS have limitations when faced with the combination of pipelined design and general hardware resource sharing. To address this issue, we propose the modular synthesis methodology, which decomposes the synthesis into compilation and linking stages, each of which captures different aspects of program structures. We show how this approach enables the pipelined resource sharing in dataflow circuits, and permits additional decoupling between control flow and data flow in the hardware, overcoming the limitations of previous approaches.

We implement our proposed techniques in PipeLink, an HLS engine that enables pipelined resource sharing in dataflow circuits. PipeLink makes a number of contributions to the dynamically scheduled HLS: (i) it proposes a modular method for synthesizing dataflow circuits, which enables fully-pipelined resource sharing in dataflow circuits; (ii) it supports both pipelined function calls and pipelined memory accesses in a unified design; (iii) it directly synthesizes efficient circuits on unmodified programs (i.e., without requiring pragmas being inserted by users). We evaluate our HLS approach against both academic and commercial HLS tools. Overall, PipeLink provides significant improvement in energy, some improvement in latency and throughput while paying some area penalty.

The remaining paper is organized as follows: Section II introduces the dataflow circuits as well as the requirements and challenges for sharing the resources at circuit level. It also discusses various options of sharing the resources and the pros and cons of existing techniques. Section III introduces PipeLink and how it leverages the modular synthesis method to enable pipelined resource sharing that is not available previously. Section IV shows the evaluation results.

II. Background

A. Dataflow circuits

Dataflow circuits consist of concurrent, independent dataflow elements. Each dataflow element communicates with the predecessors and successors through local handshake protocols, thus it does not require global control. A dataflow element remains idle until it receives input tokens from its predecessors [18]. This is different from static HLS tools, where all the building blocks require a global state machine to coordinate.

Consider the *if* example in Fig. 1 and the synthesized dataflow circuit. In the true branch, x is conditionally used

```
1  if (c) {
2    y = x + 1;
3  } else {
4    y = x * 2;
5  }
6  ... use y ...
```


Fig. 1: dataflow circuit example

```
1  y0 = f(x0);
2  ...
3  y1 = f(x1);
```
a) two invocations to f

```
1  y0 = f(x0);
2  y1 = f(x1);
```
b) sequential code

```
1  y0 = f(x0);
2  if (c) {
3    y1 = f(x1);
4  }
```
c) conditional code

```
1  y0 = f(x0);
2  do {
3    y1 = f(x1);
4  } while (c);
```
d) loop code

Fig. 2: Example code for function call

and y is redefined. In the false branch, nothing happens. After the if statement, y is used. The synthesized circuit has two types of operators: $MERGE$ and $SPLIT$ for circuit control. $MERGE$ 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. A complete set of dataflow components also includes token source, token sink, copy, function computation component, and an initial token buffer [36].

As pointed out by existing works [14], [16], [23], [28], [36], [38], dataflow circuits can be automatically pipelined. In this example, x is sent to the correct $FUNC$ based on c, and the $FUNC$ will send its result to the $MERGE$ immediately when the computation is done; the whole process is data-driven, without requiring centralized (global) control.

B. Resource sharing problem for HLS

In software programs, function calls and memory accesses (from explicitly declared arrays and dynamically allocated memories) are pervasive, and the corresponding functions and memories are shared among different callers. An HLS tool has the choice of inlining functions at each call site, or having a a single hardware block for the function that is shared by the call sites, or some combination. However, memory banks/global memory has to be viewed as a shared resource.

For the purposes of studying the resource sharing problem, 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), so we can treat memory access as a special type of function call where the function cannot be inlined/replicated. In this paper, we use "function" to denote both regular functions and memory banks, and our system supports both in a unified way. Note that the "function" refers to the regular software function, not necessarily the combinational function unit.

We are not arguing that resource sharing is the optimal solution to handle function calls and memory accesses; instead, other options such as function inlining can be used as well. However, this work focuses on pipelined resource sharing problem, so we illustrate the resource sharing design for function calls and memory accesses. It is an open question about the right balance between resource sharing and function inlining, which is out of the scope of this paper.

To start with, Fig. 2(a) shows a program segment where `f` is invoked twice. Suppose `f` is a single input, single output function, and it has two callers `y0` and `y1`. Let's annotate the two invocations to `f` as f^0 and f^1 respectively. Furthermore, let's assume `f` is not inlined. `f` is synthesized as an independent hardware block, and its in/output ports as well as `x0`, `x1`, `y0` and `y1` are synthesized as dangling ports, as illustrated in Fig. 3(a). The HLS tool needs to synthesize a resource sharing system to send `x0` and `x1` to `f` and propagates the result from `f` to `y0` and `y1` respectively.

The access order to `f` is important. Each invocation could have side-effects: `f` could modify memory (Fig. 4(a)), or update static variables (Fig. 4(b)). To guarantee correctness, it is important to make sure `f` is accessed in the same order as the original software program.

Furthermore, the number of accesses to `f` for each invocation is unknown during static synthesis stage. Instead, it needs to be dynamically computed based on runtime conditions. For example, Fig. 2(a) shows a code segment of two invocations to `f`. For sequential invocation (Fig. 2(b)), f^0 and f^1 each access `f` once. For conditional invocation (Fig. 2(c)) where c is false during a specific run, f^1 should not access `f`. For iterative invocation (Fig. 2(d)) where the loop has N iterations during a specific run, f^1 should access `f` N times before the next invocation of `f`.

To summarize, the resource sharing system needs to:

1) enforce the **access order** to `f` to be the same as specified by the programs; for example, `x0` should be sent to `f` before `x1` (Fig. 4(a));
2) count the **number of accesses** to `f` for each invocation, and starts the next invocation only when the current one has finished using `f`;

Note that because the original software program is sequential, all of the information necessary to determine the ordering and access count is available at run-time; the challenge is to compute it using dataflow operators.

C. Resource sharing options

The general resource sharing problem is illustrated in Fig. 3(a), where `f` is shared at two independent call sites. We assume `f` is a single-in-single-out resource block, x_0 and x_1 are the input tokens at two call sites, and y_0 and y_1 are the receivers to collect the results at each call site. Current HLS tools are mainly optimized for efficient synthesis of a single function, but they fall short in handling generic function calls in software programs.

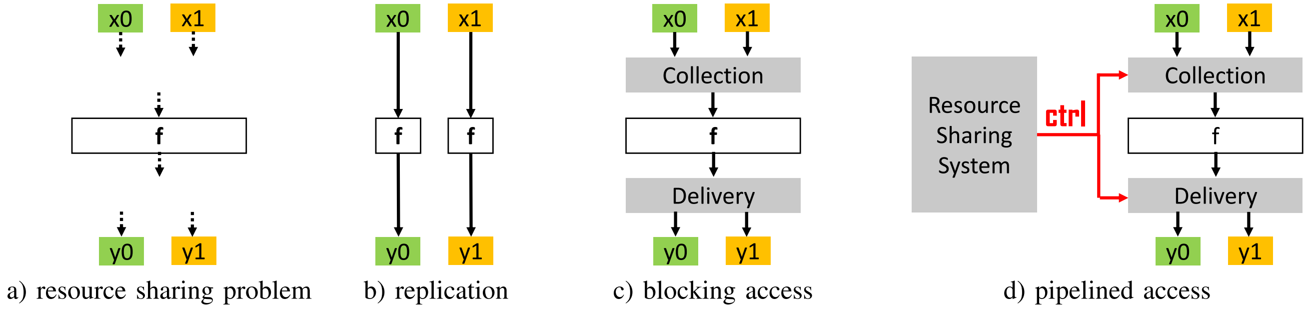


Fig. 3: *Different resource sharing options*

```
1  int A[M];
2  int f(int i) {
3    A[i]++;
4    ...
5  }
```

a) modify memory

```
1  int f(int i) {
2    static int count = 0;
3    count++;
4    ...
5  }
```

b) update static variable

Fig. 4: Side effects of the function invocation

1) *Replication*

If `f` is inlined, then the resource sharing problem is automatically solved by replicating the resources of `f` at each call site, as shown in Fig. 3(b). This is the default options for the commercial HLS tools [5], [9], [10] as well as academic HLS tools [15], [30], [32]. This solution can dramatically increase the circuit cost if the shared functions are expensive. Furthermore, this approach cannot be used for memories that are accessed from more than one program location. Some commercial tools allow users to encapsulate functions as operators and then decide how many of these to instantiate, which controls the amount of parallelism to be extracted. However, this option is not feasible when the functions have side-effects as shown in Fig. 4.

2) *Blocking sharing*

As shown in Fig. 3(c), the HLS engine should synthesize a *collection* circuit to collect and select the arguments to `f` from all callers, as well as a *delivery* circuit to deliver the results back to the appropriate caller. In the blocking approach, there is a lack of explicit *control* to coordinate the access order to `f`; hence, access to `f` is granted on a first-come first-served basis. However, the order in which access to `f` is requested *must match the software program order* whenever `f` has side-effects (e.g., Fig. 4). This means that if the underlying HLS implementation has any pipelining, extra logic is needed to guarantee that this access ordering is preserved. This applies to both accesses from different call sites, as well as repeated accesses from a fixed call site. Without extra logic, this would violate the requirements for accessing the shared resource.

To handle both these issues, existing solutions require that `f` is shared in a blocking fashion, i.e., `x1` cannot be generated and sent to `f` until `f` receives `x0` and produces `y0`. An alternate approach is to only have blocking on the collection logic end, and generate a control token in the collection logic that is used to route the results to the appropriate invocation [23]. For example, if `x0` is selected, then a control token `0` can be generated by the collection circuit and routed to the delivery logic, which then can route the result to `y0`; the *ARBITER* dataflow element can be used for this purpose. However, access to the collection circuit is still blocking. In particular, in the pipelined context, the next `x0` invocation must block until all possible future invocations from the current execution have been considered. Furthermore, using an *ARBITER* means that slack elasticity is no longer guaranteed, and hence the slack elascity feature of the circuits cannot be guaranteed.

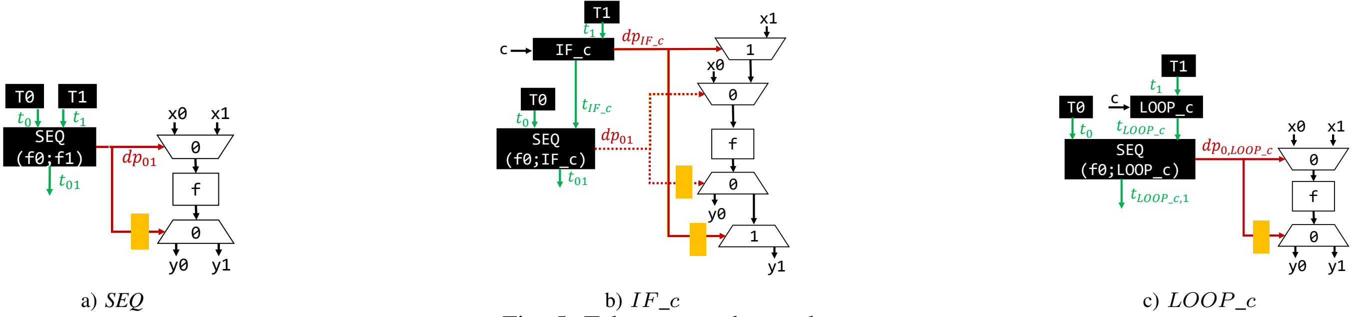
3) *Pipelined sharing*

Fig. 3(d) shows the pipelined resource sharing system for `f`, which explicitly calculates the *ctrl* tokens required. The benefit of this approach is that the control generation is *decoupled* from the data tokens, and often can be computed earlier since it only depends on the control flow of the program.

Static HLS solution. Static HLS generates a global state machine to provide central control to all operations in the circuits, so it can easily generate the desired *ctrl* tokens for the resource sharing system. However, in order to generate a pipelined hardware block for `f`, it is essential to decide the *initial interval (II)* for `f` during static synthesis stage. To do that, it is often required to inline the sub-functions and unroll all the loops inside `f` [1], [6], [10], which can result in high circuit overhead. In addition, static HLS approaches do not optimize for two common cases: (1) Conditional executions in `f`. Static HLS has to consider the worst-case scenario for synthesizing `f`. If `f` has nested, unbalanced *if/switch* statements, this problem gets worse. (2) Early-return in `f`: Sometimes a function will return in the middle of the function body for fast computation. However, the static HLS has to consider the whole function body when calculating the II, completely ignoring this shortcut.

In practice, pipelined resource sharing in static HLS is only enabled under user's discretion (typically via `#pragma` directives in C/C++) due to its potential high overhead.

Dynamically scheduled HLS solution. The dynamically scheduled HLS does not synthesize the global state machine to schedule each operator in cycles, which makes it very hard to generate the desired *ctrl* tokens to coordinate the access to the shared resource blocks. To handle this problem, we argue that the circuit synthesis should be decoupled into two stages. In the first compilation stage, each software function should be



a) *SEQ* b) *IF_c* c) *LOOP_c*

Fig. 5: Token network template

mapped into circuits independently, leaving the function calls and memory accesses unprocessed; in the second linking stage, the complete function call graph structure is analyzed, and the corresponding linking circuit is synthesized to connect the shared resources to the call sites. To our knowledge, we present the first solution to the problem of fully pipelined resource sharing for dataflow circuits that are generated from programs. Our solution results in a purely deterministic dataflow circuit implementation, ensuring slack elasticity. As a result, our approach addresses the control token generation problem in the general case [12] and supports pipelined resource sharing for dynamically scheduled HLS.

III. Moduler Synthesis Methodology

PipeLink decomposes the synthesis into two steps. During compilation step, each function is synthesized into a distinct resource block. If there are accesses to the shared resources, then the input tokens to the shared resources, together with the receivers to collect results from the shared resources, are synthesized as dangling ports, as shown in Fig. 3(a). During the linking step, PipeLink would analyze the program structure where the function call happens, and generate the control circuit to connect the dangling ports (synthesized in the compilation stage) following the program structure. The control circuit, called *Adaptive Control Token Network (ACTN)*, is itself a pipelined dataflow circuit that correctly computes the control tokens needed for pipelined resource sharing (as shown in Fig. 3(d)).

As mentioned in Section II-B, ACTN needs to count the **number of accesses** to `f` for each call site, and enforce the **access order** to the shared resource `f` as specified in programs. To achieve these, it captures the control structure of the program as well as runtime conditions to compute the control tokens. First, it generates a *use-resource* sequence for each invocation to serve as its private control token stream and resource access counter to count the number of accesses, so different invocations are decoupled from each other and do not need to compete for the single shared control token. Second, it composes different use-resource sequences to control the access order to the shared function `f`. It provides additional decoupling between control flow and data flow, reducing the chance of them blocking each other.

A. The "use-resource" sequence

A use-resource sequence is generated for each invocation to `f`. It encodes the number of accesses to `f` for the invocation during current run. It is a one-bit token stream whose value is run-length encoded. For example, if the invocation f^0 accesses `f` three times, the use-resource sequence would be `1,1,1,0`. Each "`1`" enables f^0 to access `f` once, and `0` represents the end of accessing `f` from the specific call site corresponding to the use-resource sequence. We now describe how we construct this sequence from the structure of the program.

B. The base case: a single function call

The base case corresponds to a single function call invocation. The use-resource sequence for this case is simply `1,0`, meaning this invocation would access `f` exactly once when the program reaches this call site.

C. Composing use-resource sequences

ACTN composes use-resource sequences (corresponding to all call sites of `f`) together by structural induction on the program to generate the final use-resource sequence and control tokens needed for resource sharing. There are three types: sequential, conditional and iterative.

Sequential composition: *SEQ*. Consider the example in Fig. 2(b) where two programs are composed in a sequential fashion and both include one invocation to `f`. As shown in Fig. 5(a), there have two use-resources sequences, t_0 and t_1, generated for the two invocations. Since we are constructing this by structural induction, the sequences t_0 and t_1 have variable length that in general is only known at run-time.

To compose the two programs: (a) we use *MERGE* / *SPLIT* to route the arguments/results of `f` from/to the dataflow circuits. (b) we compute the final use-resource sequence t_{01} from t_0 and t_1. (c) we generate a control token sequence dp_{01} for the *MERGE* and *SPLIT* to route the datapath tokens.

SEQ network generates t_{01} as follows: `1`s from t_0 are read and propagated to the output; when a `0` is encountered, *SEQ* switches to t_1 and continues forwarding `1`s from t_1 to the output until a `0` is encountered. Then, *SEQ* forwards the `0` to the output, and switches back to t_0. If we run the code segment in Fig. 2(b) once, t_0 is `1,0`, t_1 is `1,0` and t_{01} is `1,1,0`. dp_{01} sequence is generated as follows: for each `1` from t_0, a `0` is generated; for each `1` from t_1, a `1` is generated. The final dp_{01} is `0,1` for our example.

Conditional composition: *IF_c*. If `f` is called in a conditional statement where c is the condition, then the use-resource sequence can be computed using c. Suppose we have the following program `if(c) {S} else {T}`, and t_S and t_T

TABLE I: Normalized Performance (compared to Legup)

	PipeLink				*Commercial 1*				*Commercial 2*			
Benchmark	**Delay**	**Area**	**Energy**	**Throu.**	**Delay**	**Area**	**Energy**	**Throu.**	**Delay**	**Area**	**Energy**	**Throu.**
dfmul	0.51	1.50	0.04	1.99	1.03	0.69	0.70	0.97	1.00	1.23	0.66	1.00
dfdiv	0.53	1.40	0.01	1.89	0.47	0.88	0.39	2.11	0.72	1.93	1.67	1.39
sha	0.61	1.27	0.10	1.63	0.94	0.99	1.08	1.06	1.02	0.98	1.18	0.98
adpcm	0.49	1.29	0.02	2.08	0.86	0.55	0.82	1.17	0.75	1.04	1.16	1.33
diff	0.91	0.58	0.08	1.10	0.61	0.49	0.41	1.64	1.02	1.42	1.11	0.98
fft	0.74	1.16	0.11	1.39	0.86	0.78	0.63	1.16	0.98	0.93	0.79	1.02
fir	0.88	0.72	0.15	1.13	0.94	0.78	0.90	1.06	1.00	0.93	1.01	1.00
geomean	0.65	1.08	0.05	1.56	0.79	0.72	0.56	1.26	0.92	1.17	1.04	1.09

are the use-resource sequences for the corresponding branches. The final use-resource sequence is t_{if}, and the datapath control token sequence is dp_{if}.

If c is false we propagate `1`s from t_T to t_{if} and produces `0`s for dp_{if} until `0` is encountered. If c is true, we propagate `1`s from t_S to t_{if} and produces `1`s for dp_{if} until `0` is encountered. In the end, we appends a `0` to the end of t_{if}.

In Fig. 2(c), `f` is invoked twice: f^1 (Line 3) executed in a if block IF_C, and f^0 (Line 1) and IF_C are executed sequentially. Fig. 5(b) shows the synthesized circuits. Since f^1 is conditionally invoked, $MERGE_1$ and $SPLIT_1$ are synthesized 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 c, and two `use-resource` tokens for 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: *LOOP_c*. Suppose the use-resource sequence for the loop body is t_{body}, and the final use-resource sequence is t_{loop}. While loop condition is true, the `1`s from t_{body} are propagated to t_{loop}. When the loop terminates, we append a `0` to t_{loop}. The net effect is: t_{loop} accumulates the `1`s from t_{body} in all iterations. Since the loop has a single body, no $MERGE$/$SPLIT$ or control token generation is needed. In Fig. 2(d), `f` is invoked twice: f^0 (Line 1) executed in a sequential block, and f^1 (Line 3) executed in a loop block $LOOP_C$. Fig. 5(c) shows the synthesized circuits.

D. Summary

ACTN is dynamically constructed based on the induction of program structures using pre-designed template circuits. It operates in parallel with the datapath to decouple the control path and the data path. To reduce stalls in the use-resource network, FIFO buffers are inserted at the delivery circuit end (shown as yellow blocks in Fig. 5). As a result, new datapath control tokens can be generated (thus enabling new invocations to start) without waiting for the previous invocation to finish.

IV. Evaluation

We implement PipeLink in LLVM [7]. First, optimized LLVM IR is generated with *clang* and existing optimization passes. Next, we implement the ACTN (Section III) as LLVM passes [8]. These passes accept optimized LLVM IR as input and generate optimized dataflow circuit graphs as their output. They also perform operator clustering optimizations to group the combinational operators together for logic optimization. Finally, we build on the asynchronous circuit back-end developed by [28] to map the optimized dataflow graphs into asynchronous circuits in a 28nm technology.

A. 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 [29]. The control for each pipelined stage uses micro-pipelines [35]. The control circuits 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. We use HSPICE to extract the control circuit performance metrics, and commercial logic synthesis tool to determine delay/power/area of combinational logic, both under a 28nm process technology. Synchronous results were obtained using the same cell library and logic synthesis tool.

Comparison. We use four applications (*dfmul, dfdiv, sha, adpcm*) from an HLS benchmark suite [3], [21], and three widely used benchmarks: a differential equation solver (*differential*) from [31], and *fft* and *fir* from [15]. We compare against an academic HLS tool *LegUp* v4.0 [15] as well as two commercial HLS tools *Commercial 1* and *Commercial 2*. We use default options for all of the HLS tools and do not insert any pragmas in the benchmarks.

Metrics. We measure Delay (ps), Area (μm^2), 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.

B. Experiment results

Comparison. Table I shows the normalized performance numbers of PipeLink, *Commercial 1* and *Commercial 2* compared to *Legup*. Fig. 6 shows per-benchmark spider plots of normalized performance (relative to Legup) as well as the geometric mean of the normalized 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

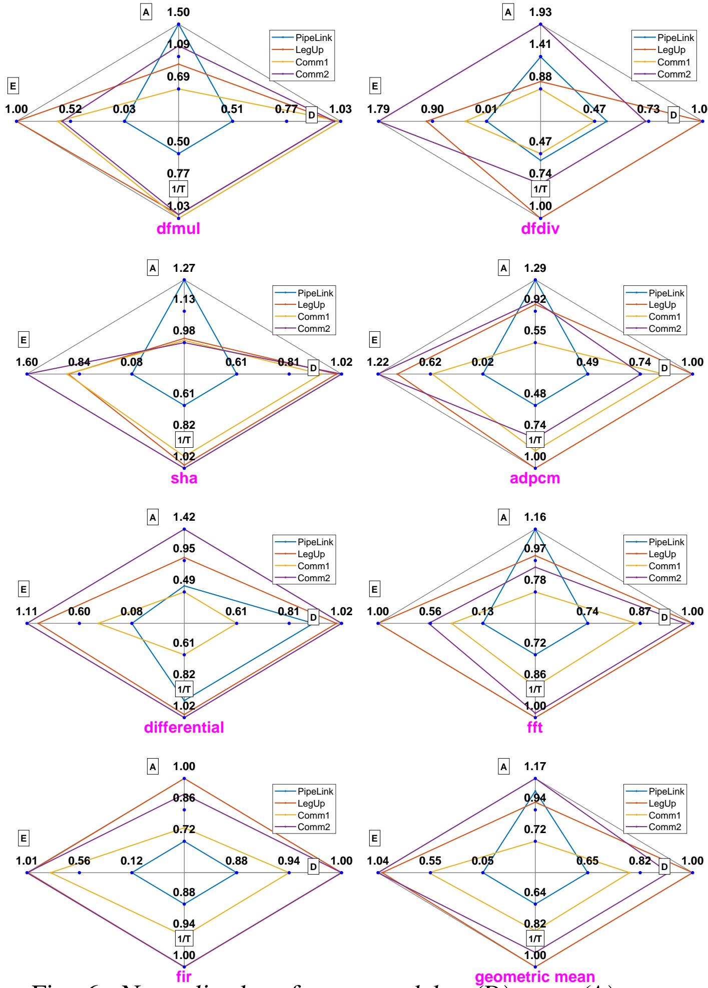


Fig. 6: *Normalized performance: delay (D), area (A), energy (E), leakage power (L), and throughput-inv (1/T)*

saving is $20X$. Even compare 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, *Commercial 1* ($0.61X$) is much faster than PipeLink ($0.91X$) because *differential* primarily consists of a loop. PipeLink has to synthesize multiple MERGEs and SPLITs for loop control [28], [36], 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. 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.

V. Related Work

Dynamically scheduled HLS. Some synchronous HLS tools generate dataflow circuits [23], [24], [37]. Lana et al. have done a series of works on mapping C/C++ programs into synchronous dataflow circuits, including [22]–[25]. DynSchedule [23], GeneralCodeSynthesis [24] and CToDataflow [25] use the distributed token network design for pipelined memory access, but the circuit pipelining could be harmed as described in Section II-C2. Furthermore, they rely on customized out-of-order LSQ [22] for pipelined memory accesses within the activated basic block in program's control flow graph, which could potentially incur high overhead. Lastly, there is not much descriptions about how they support general-purpose function calls in a pipelined manner. PipeLink however, unifies the pipelined function calls and pipelined memory accesses, and proposes an adaptive control token network design to improve circuit pipelining over distributed token network design. FunctionSynthesis [37] synthesizes functional programs into pipelined dataflow circuits. In functional programming, each function is "stateless", so their stateless functions can be shared in a pipelined manner without requiring to enforce the access order among invocations. Some tools directly map software programs into asynchronous dataflow circuits. Spatial computing [14], CASH [38] and Pegasus [13] generates asynchronous circuits which are based on the MicroPipelines [35] design, and they only support blocking function calls. AutoCtrl [20] generates distributed asynchronous control circuits automatically, but it only supports blocking function calls.

Asynchronous synthesis. Balsa [19] is based on syntax-directed translation to map message-passing hardware description language (HDL) into an asynchronous circuit, and it only allows sequential function calls. STF [36] and DataDecomposition [39] maps high-level circuit description programs into concurrent hardware modules with restricted support of pipelined function calls.

Static HLS. Most commercial tools are static HLS tools [1], [2], [4], [5], [10], and they will do function inlining by default when doing the synthesis. They also provide directives and pragmas [1], [6], [11] for users to explicitly specify function pipelining. However, pipelined resource sharing in statically-scheduled circuits could be costly for unrolling the loops and inlining functions (Section II-C3), so users need to be very careful before deciding to pipeline a function. The academia HLS tools [15], [30], [32] also share similar problems.

VI. Conclusion

We propose PipeLink, a dynamically scheduled HLS engine to map programs into asynchronous dataflow circuits. It proposes a modular synthesis methodology to support pipelined function calls and pipelined memory access uniformly. Results show that PipeLink can dramatically save energy, and improve delay and throughput performance.

REFERENCES

[1] Cadence hls. https://sen.enst.fr/system/files/c8927325ebae2ddae9620055f8a2396c/cadence_ctos_user_guide.pdf.

[2] Catapult C. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus.

[3] Chstone. https://github.com/A-T-Kristensen/patmos_HLS/tree/master/benchmarks/CHStone.

[4] Intel HLS. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[5] Legup. https://www.legupcomputing.com/.

[6] Legup directive. https://download-soc.microsemi.com/FPGA/HLS-EAP/docs/legup-9.1-docs/index.html/.

[7] LLVM. https://llvm.org/.

[8] LLVM optimization pass. https://llvm.org/docs/Passes.html.

[9] Stratus. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.

[10] Vivado hls. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf.

[11] Vivado hls directive. https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/oxu1504034430931.html.

[12] Arvind. Dataflow: passing the token. Keynote address, International Symposium on Computer Architecture (ISCA) (http://csg.csail.mit.edu/Users/arvind/ISCAfinal.pdf).

[13] Mihai Budiu and Seth Copen Goldstein. Pegasus: An efficient intermediate representation. Technical report, Carnegie Mellon University, 2002.

[14] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In Shubu Mukherjee and Kathryn S. McKinley, editors, *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, pages 14–26. ACM, 2004.

[15] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Helge Anderson, Stephen Dean Brown, and Tomasz S. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In John Wawrzynek and Katherine Compton, editors, *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, pages 33–36. ACM, 2011.

[16] Jianyi Cheng, Lana Josipovic, George A. Constantinides, Paolo Ienne, and John Wickerson. Combining dynamic & static scheduling in high-level synthesis. In Stephen Neuendorffer and Lesley Shannon, editors, *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, pages 288–298. ACM, 2020.

[17] Philippe Coussy and Adam Morawiec. *High-level synthesis*, volume 1. Springer, 2010.

[18] Jack B. Dennis. The evolution of 'static' data-flow architecture. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*. Prentice-Hall, 1991.

[19] Doug A. Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *Comput. J.*, 45(1):12–18, 2002.

[20] Euiseok Eim, Jeong-Gun Lee, and Dong-Ik Lee. Automatic process-oriented control circuit generation for asynchronous high-level synthesis. In *Proceedings Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2000) (Cat. No. PR00586)*, pages 104–113, 2000.

[21] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *International Symposium on Circuits and Systems (ISCAS 2008), 18-21 May 2008, Sheraton Seattle Hotel, Seattle, Washington, USA*, pages 1192–1195. IEEE, 2008.

[22] Lana Josipovic, Philip Brisk, and Paolo Ienne. An out-of-order load-store queue for spatial computing. *ACM Trans. Embed. Comput. Syst.*, 16(5s):125:1–125:19, 2017.

[23] Lana Josipovic, Andrea Guerrieri, and Paolo Ienne. Invited tutorial: Dynamatic: From C/C++ to dynamically scheduled circuits. In Stephen Neuendorffer and Lesley Shannon, editors, *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, pages 1–10. ACM, 2020.

[24] Lana Josipovic, Andrea Guerrieri, and Paolo Ienne. Synthesizing general-purpose code into dynamically scheduled circuits. *IEEE Circuits and Systems Magazine*, 21(2):97–118, 2021.

[25] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. From c/c++ code to high-performance dataflow circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2021.

[26] Ioannis Karageorgos, Karthik Sriram, Ján Veselý, Michael Wu, Marc Powell, David A. Borton, Rajit Manohar, and Abhishek Bhattacharjee. Hardware-software co-design for brain-computer interfaces. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, pages 391–404. IEEE, 2020.

[27] Dirk Koch, Frank Hannig, and Daniel Ziener, editors. *FPGAs for Software Programmers*. Springer, 2016.

[28] Rui Li, Lincoln Berkley, Yihang Yang, and Rajit Manohar. Fluid: An asynchronous high-level synthesis tool for complex program structures. In *27th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2021)*, 2021.

[29] Carver Mead and Lynn Conway. *Introduction to VLSI systems*. Addison-Wesley, 1980.

[30] 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, 2012*, pages 619–622. IEEE, 2012.

[31] 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.

[32] Christian Pilato and Fabrizio Ferrandi. Bambu: A modular framework for the high 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.

[33] A. Putnam et al. Chimps: A c-level compilation flow for hybrid cpu-fpga architectures. In *2008 International Conference on Field Programmable Logic and Applications*.

[34] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani 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 Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, J. P. Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin Persaud, Alex Ramírez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, 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.

[35] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 1989.

[36] John Teifel and Rajit Manohar. Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *10th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2004), 19-23 April 2004, Crete, Greece*, pages 17–27. IEEE Computer Society, 2004.

[37] Richard Townsend, Martha A. Kim, and Stephen A. Edwards. From functional programs to pipelined dataflow circuits. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 76–86, New York, NY, USA, 2017. Association for Computing Machinery.

[38] G. Venkataramani, M. Budiu, T. Chelcea, and S. Goldstein. 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*.