Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency

Licheng Guo*[1], Jason Lau*[1], Yuze Chi[1], Jie Wang[1], Cody Hao Yu[1], Zhe Chen[1], Zhiru Zhang[2], Jason Cong[1]
∗ indicates co-first authors and equal contributions
[1]University of California, Los Angeles [2]Cornell University
{lcguo,lau,cong}@cs.ucla.edu,zhiruz@cornell.edu

ABSTRACT

Designs generated by high-level synthesis (HLS) tools typically achieve a lower frequency compared to manual RTL designs. In this work, we study the timing issues in a diverse set of realistic and complex FPGA HLS designs. (1) We observe that in almost all cases the frequency degradation is caused by the broadcast structures generated by the HLS compiler. (2) We classify three major types of broadcasts in HLS-generated designs, including high-fanout data signals, pipeline flow control signals and synchronization signals for concurrent modules. (3) We reveal a number of limitations of the current HLS tools that result in those broadcast-related timing issues. (4) We propose a set of effective yet easy-to-implement approaches, including broadcast-aware scheduling, synchronization pruning, and skid-buffer-based flow control. Our experimental results show that our methods can improve the maximum frequency of a set of nine representative HLS benchmarks by 53% on average. In some cases, the frequency gain is more than 100 MHz.

1 INTRODUCTION

High-level synthesis (HLS) tools simplify the process of implementing new applications on FPGAs. They enable users to specify untimed designs in high-level languages such as C/C++ or OpenCL without concerning about cycle-accurate details at the register-transfer level (RTL). However, there still exists considerable room to improve the timing qualities of the HLS-synthesized designs. Unfortunately, current HLS tools do not provide helpful feedback or guidelines on how to improve the clock frequency at the source level or to use additional tool options. It is also challenging for regular HLS users to reverse engineer the synthesized RTL code to identify the timing bottlenecks and optimize the corresponding critical paths in the source program.

In this work, we analyze the timing issues of a diverse set of real-world HLS designs that are implemented and optimized using state-of-the-art commercial tools. To our surprise, in most cases, the frequency degradation is related to signal broadcasts. The signal broadcasts are automatically inferred or created by the HLS compiler, either in the datapath or the control logic. These broadcast structures are typically not explicitly presented in the source code, thereby often overlooked by HLS users. However, they will result in high-fanout interconnects that pose challenges to downstream physical design tools to close timing.

Here, we briefly discuss two case studies to motivate the importance of optimizing the implicit broadcasts in FPGA HLS. One is the genome sequencing accelerator [1], where we identify a data signal broadcast that sends the output of one register to tens of targets. We observe that the tool underestimates the delay of the broadcast operation, which leads to sub-optimal scheduling results. Fixing the problem boosts the final frequency from 264 MHz to 341 MHz when implemented on an Amazon F1 instance. Another example is the streaming Jacobi accelerator [2], where the pipeline control signal broadcast becomes the critical path. By optimizing the control strategy and removing its unnecessary broadcast, we improve the maximum operating frequency from 120 MHz to 253 MHz (2.1×).

Motivated by these encouraging results, we conduct a systematic analysis of the timing-critical broadcast structures in HLS, which naturally fall into two major categories:

- **Data broadcast** refers to a high-fanout signal in the datapath, which is typically formed after loop unrolling or array partitioning during the HLS compilation process. The wire delay of an operator will increase as the broadcast factor increases. Such varying delay may cause trouble to the HLS scheduler, which generally relies on static pre-characterized delay estimation.
- **Control broadcast** refers to a high-fanout control signal which typically originates from an FSM (or controller) and reaches numerous datapath components such as registers or multiplexers. In our study, we particularly focus on two critical classes of control broadcast: (1) synchronization signal broadcast and (2) pipeline control signal broadcast. These structures commonly cause timing degradation in deeply pipelined and/or highly parallelized designs. Compared to the data broadcasts, the control broadcasts are less studied in HLS.

The timing issues caused by such broadcasts are extremely hard to debug. On the one hand, data broadcast structures are hard to notice in the source code. It is difficult for HLS users to realize and understand the fact that certain "innocent-looking" software code has negative implications on the timing of the synthesized hardware. On the other hand, since most of the control signals are created by the HLS tool, they are much more challenging (if not impossible) to optimize through source code changes. Hence, a sub-optimal control broadcast may completely offset the performance gains from other sophisticated HLS optimizations. Moreover, data and control broadcasts often entangle with each other. As we will see, an HLS design as innocent as a simple buffer can suffer from both of these two broadcasts. Both data and control broadcasts must be eliminated to achieve frequency improvements.

Although the broadcasts cause serious problems in the current HLS tools, we manage to find concise and easy-to-integrate solutions. First, we use synthetic designs to capture the relationship between the increased net delay versus the broadcast factor, which serves as an effective approximation. Second, we utilize a different pipeline control methodology to trade area for a lower broadcast factor, while we further minimize the area overhead. Third, we propose to prune redundant synchronization signals to simplify the design. Our experimental results based on Vivado HLS show that (1) the timing problems caused by broadcast are indeed widespread, and (2) our proposed methods can improve the frequency of a set of representative HLS benchmarks significantly. In some cases, the gain is more than 100 MHz.

Our main technical contributions are as follows:

- We are the first to identify that the implicit signal broadcast is a major cause of the frequency degradation in highly-optimized designs synthesized using industrial-strength HLS tools. We further provide a classification of the timing-critical data and control broadcast structures.
- We propose a set of simple but effective techniques to optimize the timing of the implicit broadcasts in HLS automatically, which includes broadcast-aware scheduling, redundant synchronization pruning, and skid-buffer-based pipeline control.
- We apply our approaches to a set of nine real-world HLS benchmarks, and improve their frequency by 53% on average, with a marginal area overhead.

2 BACKGROUND

In a typical HLS flow, the *scheduling* phase inserts clock boundaries into the original untimed specification. Generally, the HLS scheduler performs a high-level estimation of the operation delay. This estimation is usually based on pre-characterized statistics, which include the delay of common components for computing, storage, and interconnects (e.g., adders, multipliers, registers, BRAMs, multiplexers, etc). However, HLS tools lack consideration of the additional net delay in broadcast structures. The predicted delay by HLS tools for a certain operator is fixed regardless of the actual environment. Thus, the actual value for operators near broadcasts is usually larger than the predicted value.

The *RTL generation* phase creates the control logic to orchestrate the datapath, controlling each stage to execute at its scheduled cycle. For fully-pipelined [3] datapath, the `enable` signals for activation or the `stall` signals for flow control will be broadcast to every element of the pipeline to operate the datapath as a whole. Meanwhile, the FSM proceeds to the next stage only when all concurrent modules at the current stage signal their completion to the controller. This aggregated condition of `done`s is used as the next `start` signal, and will be broadcast to the parallel modules in the next stage. The same logical functionality can be mapped to different implementations, thus it is important that the selected implementation is efficient and scalable, otherwise this step may offset the gains of all other intricate optimizations.

3 CLASSIFICATION OF HLS BROADCASTS

3.1 Data Signal Broadcast

By implicit data broadcast, we broadly refer to signal broadcasts in the HLS-synthesized datapath. These broadcasts are specified by the source code and directives, though they are less obvious to the users. As is explained in Section 2, current HLS tools have a fixed delay estimation for a certain operator. However, such estimation is no longer accurate with large data broadcasts.

We create two synthetic examples of common HLS design patterns to show how data signal broadcast structures are formed, and what limitations in current HLS tools lead to this problem:

1) Loop unrolling, as in Figure 1. The variable `source` is defined outside the loop body, and is loop-invariant. Since it is accessed in each iteration, in the corresponding hardware shown in Figure 2, the register for `source` is connected to 1024 instances of the loop body, resulting in a data signal broadcast.

Obviously in this case, the actual delay of the `add` operator in `"source + foo"` (line 5) includes the additional wire delay between the `source` register and the `add` operators. However, the current HLS delay model does not consider such a broadcast cost. Therefore, the scheduler still views the delay of this 1024-broadcast-add the same as a normal `add` without broadcast.

For example, in Figure 2, assume the delay of a simple `add` or `sub` operator is 1.5ns, while the actual delay for the 1024-broadcast-add is 2.5ns. If the timing target is 3ns, the HLS tool will schedule the `add` and `sub` to be performed within the same cycle, while they should have been separated to meet the timing constraint.

2) Large buffer and memory arrays, as in Figure 3. On FPGAs, a large on-chip buffer will be implemented as multiple block RAMs (BRAMs). Thus, the data `source` will fan out to many physically scattered memories, though they jointly form a single logical entity.

When the buffer size increases, the `load` and `store` operations will also suffer extra wire delays. However, most existing HLS tools do not take them into consideration either. The predicted delay remains the same regardless of the size of the buffer. This results in inadequate pipelining between the BRAM units and the data source/sink. For example, the double-buffer technique requires distributing data to the local buffers of multiple parallel processing elements (PEs) [4], which tend to be inadequately pipelined; the HLS support for dynamic data structures also requires large buffers [5], where their accesses degrades the maximum frequency.

```
1 data_t source = ...;           // loop-invariant variable
2 for (size_t i = 0; i < 1024; i++) {
3 #pragma HLS unroll
4   foo = ...i...; bar = ...i...;  // loop-dependent
5   dest[i] = source + foo - bar; /* ... */ }
```

Figure 1: Code of data broadcast - Example #1: loop unrolling.

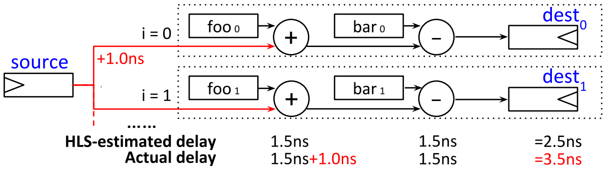


Figure 2: HLS-generated architecture of Fig. 1

```
data_t buffer[737280]; // mapped to multiple BRAM units
buffer[idx] = source; // `source` connects to every BRAM unit
```

Figure 3: Code of data broadcast - Example #2: large array.

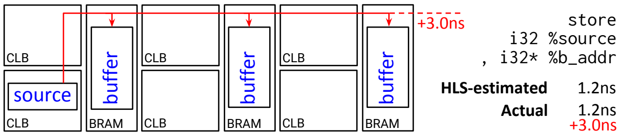


Figure 4: HLS-generated architecture of Fig. 3

3.2 Control Signal Broadcast - Synchronization

The synchronization logic originates from the parallelization of the sequential source code. The HLS scheduler automatically infers parallelism and schedules independent functions and operations to the same state for concurrent execution. To guarantee correctness, the HLS strictly follows the original semantics and generates control logic to wait for all parallel modules to complete before proceeding to the following operations. However, this fixed synchronization template may not be optimal for large designs and may introduce critical paths if the degree of parallelism scales up.

Figure 5a shows one scenario of this broadcast. For streaming designs, users describe the dataflow graph in sequential C++ code and let the HLS tools infer the parallelism. However, if multiple streaming kernels are defined in the same loop, the HLS will pedantically synchronize them at the granularity of one iteration. As a result, independent flows are glued together and form a broadcast of the synchronization signal. Figure 6a visualizes this situation.

Figure 5b shows another example, where multiple independent instances of `PE_*()` are called, and they execute in parallel. The controller waits for all of them to finish, then reads their outputs together and proceeds to the next FSM stage. Figure 6b shows the corresponding logic generated by HLS.

Although functionally correct, such synchronization strategy is not scalable. The complexity of routing such “reduce-broadcast” signals will soon explode with increasing degrees of parallelism. Optimization of the synchronization logic is necessary.

3.3 Control Signal Broadcast - Pipeline

For a pipeline that interacts with modules with flow control interfaces (e.g., FIFOs), the most common approach of current HLS tools is to broadcast the back-pressure signals (e.g., `empty`/`full` and `valid`/`ready`) to control the flow. Figure 7 shows an example, and Figure 8 shows the corresponding inferred broadcast structure.

Previous works based on the theory of latency-insensitive design analyze the role of back-pressure in a theoretical way [6], but lack consideration in the aspect of circuit implementation. Though effective for small designs, such methodology will soon become the critical path with increasing pipeline sizes.

4 APPROACHES

4.1 Broadcast-Aware Scheduling

As previously mentioned, the current HLS delay estimation does not consider the extra wire delay caused by the broadcast. Decades of research on HLS have shown that it is extremely hard to have an accurate delay estimation without the placement information [7]. However, here we propose a simple but effective method can be used to approximate this extra delay.

We implement skeleton broadcast structures on an empty FPGA to obtain the post-routed delay. For example, in one skeleton design, we instantiate 64 adders, and one of the two input ports of every adder is connected to a common source register. For buffer access operations (`load`, `store`), we record the actual delays of different buffer sizes. In this way, we collect reusable statistics of calibrated delays for each combination of operator, data type and broadcast factor. Each data point is averaged with its neighbors to suppress random noise caused by the heuristic optimization in downstream processes. When the broadcast factor is small, the delay obtained from our experiment is consistent with the predicted delay of the Vivado HLS tool. In fact, current HLS tools adopt a similar pre-characterize approach to build up their delay model, except that they do not characterize the effects of broadcasts as we do.

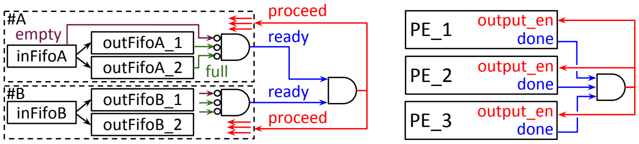
```
#pragma HLS dataflow
while (1) {
  /* --- inferred parallelization --- */
  inFifoA.read(&a);
    outFifoA1.write(a.foo); outFifoA2.write(a.bar); // #A
  inFifoB.read(&b);
    outFifoB1.write(b.foo); outFifoB2.write(b.bar); // #B
  /* --- HLS infers excessive synchronization --- */ }
```

(a) Code of synchronization - Example #1.

```
data_t kernel( ...... ) {
  /* --- inferred parallelization --- */
  aOut = PE_1(aIn); bOut = PE_2(bIn); cOut = PE_3(cIn); // ...
  /* --- inferred synchronization --- */
  return aOut + bOut + cOut /* ... */; }
```

(b) Code of synchronization - Example #2.

Figure 5: Example code of sync among parallel modules.



(a) Architecture of Example #1 (b) Architecture of Example #2

Figure 6: HLS-generated architecture of the code above.

```
for (int i = 0; i < ITER; i++) {
#pragma HLS pipeline
  input_fifo.read(&a); /* implicit "empty"-based stall */
  b = inlined_datapath_foo(a);
  output_fifo.write(b); /* implicit "full"-based stall */ }
```

Figure 7: Code example of pipeline control signal broadcast.

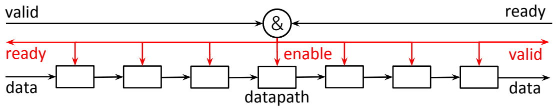
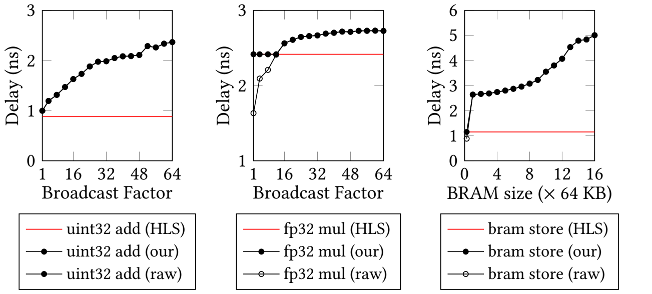


Figure 8: HLS-generated architecture of pipeline control.

Figure 9 shows our measured delay of the add operation and the BRAM buffer access of the `int` type, and multiplication of the `float` type by Xilinx Vivado. For the add and buffer access operations, the delay values obtained by our experiments perfectly match with the Vivado-HLS-predicted values when the broadcast factor is small. For large broadcast factors, our measurement significantly surpasses the HLS-predicted values, which reveals the inaccuracy of current delay estimations under large broadcast factors. For the multiplication, the HLS-predicted delays are much higher than that in our experiments, possibly because the Vivado HLS tool is being deliberately conservative about multiplication for floating points. Therefore, we choose the maximum between the HLS-predicted delay and our experimented results as our calibrated delay.



(a) 32 bits integers adder (b) 32 bits FP multiplier (c) BRAM buffer access

Figure 9: Vivaod HLS estimated delay, our calibrated delay and raw experimental delay on different operators.

To inject our calibration into the HLS tool, we parse the HLS scheduling reports, which include the LLVM instructions annotated with scheduled state/cycle, estimated delay, etc. For arithmetic operations, we analyze all RAW dependencies to obtain the broadcast factor (how many times a variable is read by later instructions in the same cycle) and then calculate the calibrated total latency for each chain of operations that are scheduled within one cycle. For detected violations of the target frequency, we insert register modules to the source code, which is equivalent to forcing the scheduler to split the operations into different cycles. If a broadcast of floating-point multiplication by itself surpasses the delay target, we also add additionally pipelining to facilitate downstream retiming. For memory operations, the predicted delay is adjusted based on the size of the buffer allocated, and additional pipelining will be added to variables interacting with the buffer at the source code. For memory access to large buffers within a pipelined environment, we are safe to add additional latency as this will not change the pipeline initiation interval.

Another potential option is to explicitly construct a broadcast tree in the source code to deal with huge broadcasts. However, it is difficult to model the influence of different tree topologies on the black-box physical design process. Our extensive experimental experiences also show that it is better to let the physical design tools handle the register duplication during placement, in which phase the delay model and knowledge of layout are more comprehensive and accurate.

4.2 Synchronization Logic Pruning

The redundant synchronization logic may severely limit the maximum achievable frequency. From the perspective of the downstream logic synthesis tools, they cannot be optimized away; but with high-level information, we are able to identify and get rid of these synchronizations.

For the first case—dataflow synchronization, as shown in Fig. 5a, we propose to isolate the independent flow paths in the flow graph. We reconstruct the dataflow graph, not based on the user-defined streaming kernels, but at the granularity of the elementary flow control units. We identify the isolated sub-graphs within user-defined streaming kernels and split the independent flows explicitly into separate loops, which avoids the unwanted synchronization from the HLS compiler. Figure 10a shows the optimized logic.

For the second case — synchronizing parallel modules, as shown in Figure 5b, the key idea is to only wait for the part with the longest latency (Figure 10b). We read the HLS schedule report to look for modules with determined latency for synchronization pruning. Our method cannot handle modules with dynamic latency, but it is possible to adopt symbolic execution to handle more situations, for example loops with variable bounds. This remains our future work.

4.3 Skid-Buffer-Based Pipeline Control

We identify that the flow control broadcast can be avoided by a common practice of adopting additional *bounded-size* buffering called a skid buffer [8], which is shown in Figure 11. We further improve this method by minimizing the area overhead.

Instead of switching the whole pipeline between two modes—active and stalled, we transform the control logic to keep the pipeline *always flowing*, and associate a `valid` bit with each data. The key to avoiding overflow is the skid buffer (an extra bypass FIFO) appended at the end of the pipeline. When the downstream is not ready, data will accumulate in the buffer. Then the buffer will become non-empty, and the pipeline will stop reading from the upstream, so the later pipeline inputs will be invalid bubbles. Assuming the length of the pipeline is N, as long as the depth of the buffer is no smaller than $N + 1$ (+1 since the `empty` signal will be deasserted one cycle after the first element is in), no overflow will happen. We refer to this practice as *skid-buffer-based pipeline control*. Note that this approach has the exact same throughput as the original stall-based back-pressure control.

However, this method introduces area overhead. For the original implementation, the area overhead will be:

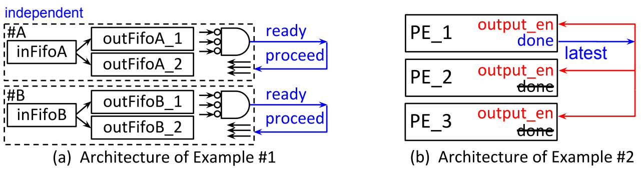
$$BufferArea = (N + 1) \cdot w_\beta$$

where N is the depth of the pipeline and w_β is the width of the output data of stage β, as marked in Figure 11.

Observe that the skid buffer can be split and distributed into the datapath, as shown in Figure 12. Instead of an N-depth buffer of width w_β at the end of the whole pipeline, we can insert an (M+1)-depth buffer of width w_α after the M-th stage, and an (N-M+1)-depth buffer of width w_β after the final stage.

The new area overhead will be:

$$BufferArea' = (M + 1) \cdot w_\alpha + (N - M + 1) \cdot w_\beta$$



(a) Architecture of Example #1 (b) Architecture of Example #2

Figure 10: Pruned architecture corresponding to Figure 6.

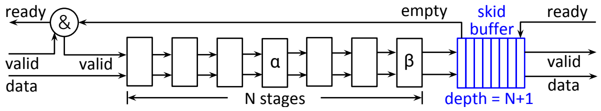


Figure 11: Skid-buffer-based pipeline control architecture.

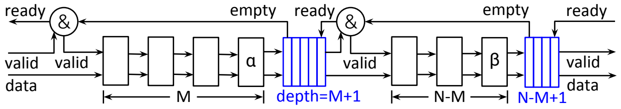


Figure 12: Multi-level skid-buffer-based pipeline control.

The minimization of the buffer area can be easily solved using dynamic programming, and the details are omitted here.

To obtain the data width between stages, we parse the schedule report and collect the definition location and usage location for each variable, thus obtaining the total data width passed between stages. Due to the engineering challenges to modify the HLS tool without access to the source code, our proof-of-concept implementation of the proposed solutions still involves some manual part. We hope that the identified issues and corresponding solutions appeal to HLS vendors and can be integrated into their commercial tools.

5 EXPERIMENTS

Our experiments are based on the Vivado HLS since most open-source HLS benchmarks are developed using it. We use the Vivado version 2018.2 with default mode. Retiming and fan-out optimization are enabled. The target FPGA chips are based on the choices of the original sources of the designs.

5.1 Benchmarks

Our results are in Table 1. The genome sequencing design is from [1]. We adjust the broadcast factor by changing `BACK_SEARCH_COUNT`. The LSTM inference network design is from [9]. We adapt the `HLS_N-Node` part, change the data type to floating point and set N to be 256. The face detection design is from the Rosetta benchmark [11]. The matrix multiply and the pattern matching design are adapted from [4]. We further increase the parallelism of the matrix multiplication design to expose the problem. The Jacobi stencil kernel and its HBM version are generated by the SODA compiler [2]. The streaming buffer design consists of two loops, which first write to a very large buffer and then read from the buffer.

5.2 Broadcast-Aware Scheduling

We illustrate the experiment with [1] in detail as a case study to present our broadcast-aware scheduling method.

```
#pragma HLS pipeline
#define UNROLL_FACTOR 64
// ......
for (int j = 0; j < UNROLL_FACTOR; j++) {
#pragma HLS unroll
    dist_x = prev[j].x - curr.x;
    dist_y = prev[j].y - curr.y;

    dd = dist_x > dist_y ? dist_x - dist_y : dist_y - dist_x;
    min_d = dist_y < dist_x ? dist_y : dist_x;
    log_dd = log2(dd); // a series of if-else
    temp = min_d > prev[j].w ? prev[j].w : min_d;

    dp_score[j]= temp - dd * avg_qspan - (log_dd>>1)
    if((dist_x == 0 || dist_x > max_dist_x )||
        (dist_y > max_dist_y || dist_y <= 0) ||
        (dd > bw) || (curr.tag != prev[j].tag) ){
        dp_score[j] = NEG_INF_SCORE;
}   } ......
```

Figure 13: Design code snippet from [1]. The loop-invariant variables (broadcast sources) are marked blue.

Figure 14 shows an operation chain scheduled by HLS. Since `curr.x` is consumed by 64 `sub` operators, in comparison to the HLS predicted delay, we adjust the predicted delay of the `sub` from 0.78ns to 2.08ns according to our measurement of the skeleton designs. Therefore, we insert a register module to force the splitting of the operation chain. Figure 15b shows the frequency gain.

Table 1: Timing improvements and post-implementation resources on HLS designs using our proposed solutions.

Application	Broadcast type	Target FPGA	LUT (%)		FF (%)		BRAM (%)		DSP (%)		Freq (MHz)		
			Orig	Opt	Orig	Opt	Orig	Opt	Orig	Opt	Orig	Opt	Diff
Genome Sequencing [1]	Data	UltraScale+ (AWS F1)	22	22	11	12	6	6	8	8	264	341	29%
LSTM Network [9]	Data	UltraScale+ (AWS F1)	8	9	6	6	2	2	14	14	285	325	14%
Face Detection [10]	Data	ZYNQ (ZC706)	21	22	14	15	16	16	9	9	220	273	24%
Matrix Multiply	Pipe. Ctrl. & Data	UltraScale+ (AWS F1)	23	23	24	27	25	25	74	74	202	299	48%
Stream Buffer	Pipe. Ctrl. & Data	UltraScale+ (AWS F1)	1	1	1	1	95	95	0	0	154	281	82%
Stencil [2]	Pipe. Ctrl.	UltraScale+ (AWS F1)	40	40	41	41	30	29	83	83	120	253	111%
Vector Arithmetic	Pipe. Ctrl. & Sync.	UltraScale+ (AWS F1)	17	17	16	15	0	<1	60	60	195	301	54%
HBM-Based Stencil [2]	Pipe. Ctrl. & Sync.	UltraScale+ (Alveo U50)	21	23	23	23	34	31	37	37	191	324	70%
Pattern Matching [4]	Data & Sync.	Virtex-7 (Alpha-Data)	17	17	5	7	9	9	0	0	187	278	49%

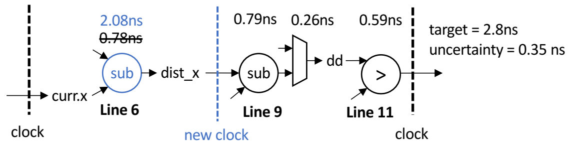
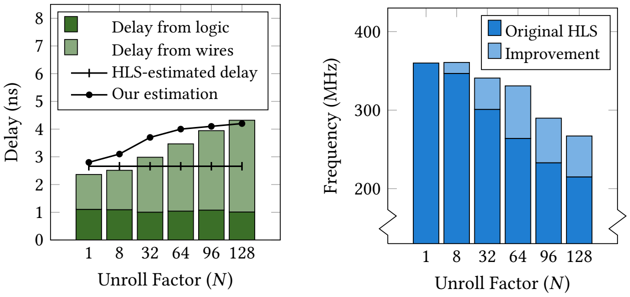


Figure 14: An operation chain with broadcast operators.



(a) The delay estimations of HLS and our tool, and the actual delay of a critical path in the original [1] design.

(b) Achieved frequency of the [1] design using HLS's original schedule and our schedule on different unroll factors.

Figure 15: Optimization of data broadcast.

Experiment results show that our method approximates the actual delay in a more reasonable way, while the HLS-estimated delay is invariant to broadcast factors. Although our result does not match the actual perfectly, our frequency gain shows that this is helpful for broadcast operations, since less neighboring logic will be put in the same cycle of the broadcast, which will facilitate downstream retiming and fanout optimization. As for overhead, the length of the pipeline is 9 originally and 10 after optimization. Both have the same initiation interval of 1. There will be fairly small overhead in the usage of flip-flop, which is generally negligible.

5.3 Synchronization Logic Pruning

We use the HBM-based (High-Bandwidth Memory) Jacobi stencil acceleration kernel generated by the SODA compiler [2, 12], which uses 28 independent memory ports of the HBM. The 512-bit data from each HBM port is scattered into 8 64-bit FIFOs, later to different streaming kernels. However, the SODA compiler expresses the 28 independent flows together in a single loop, forming a sync broadcast pattern similar to Figure 6a. Thus there is a synchronization among all HBM ports and all destination FIFOs. We prune the unnecessary sync by splitting the independent parts into different loops. This boosts the frequency from 191 MHz to 324 MHz.

5.4 Skid-Buffer-Based Pipeline Control

We again experiment with the SODA compiler [2], but this time generate the 2D Jacobi kernel as a whole pipeline. We concatenate different iterations of the kernel to change the size of the pipeline.

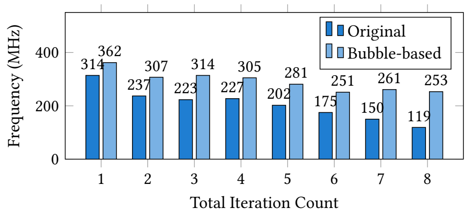


Figure 16: The achieved frequency of the Jacobi kernels.

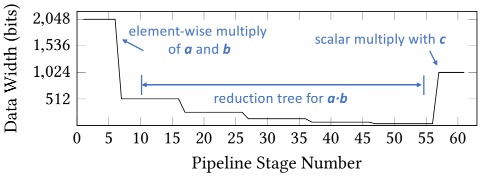


Figure 17: Bitwidth of the passed data between stages.

Table 2: Experiment results on 512-wide vector product.

Implementation	Frequency	LUT	FF	BRAM	DSP
Stall	195 MHz	17%	16%	0%	60%
Skid Buffer	299 MHz	18%	16%	12%	60%
Min-Area Skid Buf.	301 MHz	17%	15%	0.02%	60%

Each iteration takes about 5% of LUT, 5% of Flip-Flop, 4% of BRAM and 10% of DSP. Figure 16 shows the improvements by changing the pipeline control logic to the skid-buffer-based method. For the super pipeline of eight Jacobi iterations, it has 370 datapath stages and produces 512-bit results. Since this pipeline has a spindle shape, the best strategy is to add the entire buffer at the end of the pipeline. The corresponding buffer only costs about 23KB of BRAM resource.

We present a synthetic example to further demonstrate the benefit of our dynamic programming algorithm to minimize the area of the extra buffer. Assume the pipeline computes $(\mathbf{a} \cdot \mathbf{b})\mathbf{c}$, where the dot-product of vector **a** and **b** is scalar-multiplied with vector **c**. A reduction tree is inferred for $\mathbf{a} \cdot \mathbf{b}$, and the output scalar is multiplied with **c**. Figure 17 shows the case for 32-wide vector of float numbers. Note that in stage #56 only one number (result of $\mathbf{a} \cdot \mathbf{b}$) is passed through. Thus, the first stages #1 to #56 should be buffered separately from the stages after #56. Directly adding a buffer at the end results in $(61 + 1) \times 1024 = 63488$ bits while the optimized version costs $(56 + 1) \times 32 + (5 + 1) \times 1024 = 7968$ bits. Table 2 shows the results for the 512-wide vector product.

5.5 Combined Effect

In many real-world cases, we must combine these two aforementioned approaches to truly resolve the timing degradation. For example, Figure 18 shows a simple stream buffer with both data and control broadcasts. The source data register is connected to each of the BRAM units, forming an implicit data broadcast. Besides, the `enable` back-pressure signal is broadcast to all BRAM units.

```
loop1: for (int i = 0; i < BIG_SIZE; i++) {
#pragma HLS pipeline II=1
   in_fifo.read(&buffer[i]); } // data into buffer
loop2: for (...) ... // data out of buffer
```

Figure 18: Code for the large buffer access example.

Based on the size of the array and the pipeline environment, additional latency is added to optimize the data broadcast. Meanwhile, the skid-buffer-based pipeline control is used to avoid the broadcast of `enable`. Figure 19 shows the achieved frequency of varying buffer sizes. Three batches of experiments are done: the original one; the version which only has the data broadcast optimized; the version with both the data and control broadcast optimized. As is obvious, we need to optimize both the data broadcast and the control broadcast to achieve scalable performance.

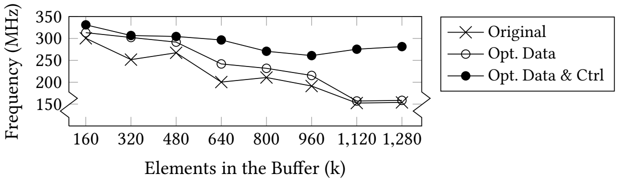


Figure 19: Achieved frequencies of the stream buffer design.

Another example is pattern matching from [4] with both data and sync control broadcast similar to Figure 6b. Addressing both of them lead to a high frequency boost, as in Table 3.

Table 3: Experiment results on pattern matching.

Implementation	Frequency	LUT	FF	BRAM	DSP
Original	187 MHz	17%	5%	9%	0%
Opt. Data	208 MHz	18%	7%	9%	0%
Opt. Data & Ctrl	278 MHz	17%	7%	9%	0%

6 RELATED WORK

General high-fanout optimization. fanout optimization has been extensively studied in logic-synthesis [13, 14, 15] and physical design [16, 17, 18, 19]. However, optimization approaches at these levels are restricted by the cycle-accurate timing specification of the RTL input. For example, they cannot arbitrarily divide the broadcast delay into two or more clock cycles; retiming [19, 20] will not work without enough registers on the path, etc. In contrast, optimization at the behavior level is more effective as we can change the schedule of the broadcast. Even though the original designs are implemented with modern backend broadcast optimization, our behavior-level optimizations still bring huge frequency gain. Cong *et al.* [21] uses a multi-level broadcast tree for the control signal, but needs iterative tuning for a satisfying tree topology.

Physically aware optimization for HLS. Zhang *et al.* [7] propose to iteratively run placement and routing to calibrate the delay information used by HLS. However, this approach requires tens of hours of compilation time overhead for each design. Meanwhile, even with the accurate delay, it still cannot address the timing issue caused by the auto-inferred control logic. Zhao *et al.* [22] and Tan *et al.* [23] show that the delay prediction of logic operations (e.g., `AND`, `OR`, `NOT`, etc) is too conservative, so they propose to consider the technology mapping for logic operations. Fujiwara *et al.* [24, 25] try to model clock skew at the behavior level. All of these efforts are orthogonal to our work. In contrast, we propose ways to calibrate the delay prediction for broadcasts. Cong *et al.* [26] propose preliminary metrics to evaluate the layout-friendliness of an RTL netlist, which lacks consistent accuracy. Tatsuoka *et al.* [27, 28] reports which lines of source codes will lead to MUX and deMUX.

Optimization of HLS data broadcast. Cong *et al.* [29] tried to solve a specific case of the data broadcast problem in our classification. They attempted to alleviate the critical path by accessing large buffers, which may be mapped to scattered BRAM units. However, they require explicit user intervention and iterative tuning to explore the best topology. They do not consider the ultimate limitation of the HLS. Moreover, they can only re-arrange the data interconnect between the external port and each explicitly-defined processing element, but not fine-grained datapath. However, their approach is suboptimal compared to our data-control co-optimization. 1) We add more pipelining between the data port and target buffers. 2) we optimize the corresponding improper control logic. Besides, we avoid user intervention and iterative tuning.

7 CONCLUSION

In this paper, we analyze the common types of broadcast in HLS. We present delay model calibration, synchronization pruning and min-area skid-buffer-based pipeline control. We bring over 50% of frequency gain on real-world designs.

8 ACKNOWLEDGEMENT

This work was supported by the CRISP center under the JUMP program, the NSF/Intel CAPA Program, the NSF NeuroNex Program DBI-1707408 and CDSC industrial partners Huawei, Mentor, Xilinx.

REFERENCES

[1] L. Guo et al. Hardware acceleration of long read pairwise overlapping in genome sequencing: a race between FPGA and GPU. *FCCM '19.*

[2] Y. Chi et al. SODA: stencil with optimized dataflow architecture. *ICCAD '18.*

[3] P. Zhou et al. Energy efficiency of full pipelining: a case study for matrix multiplication. *FCCM '16.*

[4] J. Cong et al. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. *DAC '18.*

[5] J. Lau et al. HeteroRefactor: refactoring for heterogeneous computing with FPGA. *ICSE '20.*

[6] L. P. Carloni. The role of back-pressure in implementing latency-insensitive systems. *Electronic Notes in Theoretical Computer Science*, 146(2):61–80, 2006.

[7] H. Zheng et al. Fast and effective placement and routing directed high-level synthesis for FPGAs. *FPGA '14.*

[8] Intel. *Intel Hyperflex Architecture High-Performance Design Handbook*, page 57.

[9] Z. Chen et al. Clink: compact LSTM inference kernel for energy efficient neurofeedback devices. *ISLPED'18*, page 2.

[10] N. K. Srivastava et al. Accelerating face detection on programmable SoC using C-based synthesis. *FPGA '17.*

[11] Y. Zhou et al. Rosetta: a realistic high-level synthesis benchmark suite for software programmable FPGAs. *FPGA '18.*

[12] Y. Chi et al. Exploiting computation reuse for stencil accelerators. *DAC '20.*

[13] M. Pedram et al. Layout driven logic restructuring/decomposition. *ICCAD'91.*

[14] H. J. Hoover et al. Bounding fan-out in logical networks. *JACM*, volume 31 of number 1, pages 13–18.

[15] K. J. Singh et al. A heuristic algorithm for the fanout problem. *DAC '90.*

[16] T. Okamoto et al. Buffered steiner tree construction with wire sizing for interconnect layout optimization. *ICCAD '96.*

[17] G. Beraudo et al. Timing optimization of FPGA placements by logic replication. *DAC '03.*

[18] N. Weaver et al. Post-placement c-slow retiming for the Xilinx Virtex FPGA. *FPGA '03.*

[19] N. Weaver. *Retiming, repipelining and c-slow retiming.* 2008, pages 383–399.

[20] B. Van Antwerpen et al. Register retiming technique. US Patent 7,120,883.

[21] J. Cong et al. SMEM++: a pipelined and time-multiplexed SMEM seeding accelerator for genome sequencing. *FPL '18.*

[22] R. Zhao et al. Area-efficient pipelining for FPGA-targeted high-level synthesis. *DAC '15.*

[23] M. Tan et al. Mapping-aware constrained scheduling for LUT-based FPGAs. *FPGA '15.*

[24] K. Fujiwara et al. Clock skew estimate modeling for FPGA high-level synthesis and its application. *ASICON '15.*

[25] K. Fujiwara et al. A high-level synthesis algorithm for FPGA designs optimizing critical path with interconnection-delay and clock-skew consideration. *VLSI-DAT '16.*

[26] J. Cong et al. Towards layout-friendly high-level synthesis. *ISPD '12.*

[27] M. Tatsuoka et al. Physically aware high level synthesis design flow. *DAC '15.*

[28] M. Tatsuoka et al. Wire congestion aware high level synthesis flow with source code compiler. *ICICDT'18.*

[29] J. Cong et al. Latte: locality aware transformation for high-level synthesis. *FCCM '18.*