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

Licheng Guo*, Jason Lau*, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang and Jason Cong
University of California Los Angeles, Cornell University

* indicates co-first authors

https://github.com/Licheng-Guo/vivado-hls-broadcast-optimization
Outline

• Introduction
• Problem Classification
• Solution
• Experiments
RTL Verilog vs. Untimed C/C++

- Much higher developing efficiency
- Less achievable frequency compared to RTL designs
- Hard to debug the critical path

```verilog
module dut(rst, clk, q);
  input rst;
  input clk;
  output q;
  reg [7:0] c;
  always @(posedge clk)
    begin
      if (rst == 1b1) begin
        c <= 8'b00000000;
      end
      else begin
        c <= c + 1;
      end
    end
  assign q = c;
endmodule
```

```c
uint8 dut() {
  static uint8 c;
  c++ = 1;
}
```

An 8-bit counter

High-Level Synthesis

Google Scholar: vivado hls fpga
Articles
About 11,300 results (0.10 sec)

Google Scholar: intel opencl fpga
Articles
About 8,310 results (0.04 sec)
We Analyze the Timing Issues of Complex Designs
We Analyze the Timing Issues of Complex Designs

• Most critical paths are related to broadcasts
  • Some are hidden in user codes
  • Some are inferred by the HLS compiler
  • Lead to high-fanout interconnects and bad timing quality
We Analyze the Timing Issues of Complex Designs

- Most critical paths are related to broadcasts
  - Some are hidden in user codes
  - Some are inferred by the HLS compiler
  - Lead to high-fanout interconnects and bad timing quality
- We categorize common types of broadcasts in HLS-based designs.
We Analyze the Timing Issues of Complex Designs

• Most critical paths are related to broadcasts
  • Some are hidden in user codes
  • Some are inferred by the HLS compiler
  • Lead to high-fanout interconnects and bad timing quality

• We categorize common types of broadcasts in HLS-based designs.

• We analyze the inherent limitations of current HLS tools exposed by the broadcast problem
We Analyze the Timing Issues of Complex Designs

• Most critical paths are related to broadcasts
  • Some are hidden in user codes
  • Some are inferred by the HLS compiler
    • Lead to high-fanout interconnects and bad timing quality

• We categorize common types of broadcasts in HLS-based designs.

• We analyze the inherent limitations of current HLS tools exposed by the broadcast problem

• Our lightweight solutions bring significant frequency boost on real-world HLS designs
Outline

• Introduction
• Problem Classification
• Solution
• Experiments
Classification of Broadcasts

• Data Broadcast
  • Originate from the source code
  • High fan-out signals in the datapath
  • Can be mapped back to certain lines in the source code
Classification of Broadcasts

• Data Broadcast
  • Originate from the source code
  • High fan-out signals in the datapath
  • Can be mapped back to certain lines in the source code

• Control Broadcast
  • Originate from the compiler
  • High fan-out signals from control logic
  • Completely transparent to users
Data Broadcast

• Scenario 1: unrolled loop

```c
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; /* ... */
```
Data Broadcast

• Scenario 1: unrolled loop

```c
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; /* ... */
6 }
```
Data Broadcast

• Scenario 1: unrolled loop

```c
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; /* ... */
}
```
Data Broadcast

• Scenario 1: unrolled loop

```c
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; /* ... */
```

Problem: current HLS delay model does not consider the additional net delay
Data Broadcast

• Scenario 1: unrolled loop

```c
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; /* ... */
}
```

underestimated delay --> inadequate registering
Data Broadcast

• Scenario 2: Large buffer

```c
1 data_t buffer[737280]; // mapped to multiple BRAM units
2 buffer[idx] = source; // `source` connects to every BRAM unit
```
Control Broadcast

• Scenario 1: Pipeline backpressure

```c
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 */
}
```
Control Broadcast

• Scenario 1: Pipeline backpressure

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

- **Scenario 2: Synchronization of parallel logics**
- The compiler infers parallelism from sequential code
- Insert synchronization logic to guarantee correctness

```c
1 data_t kernel( ...... ) {
2  /* --- inferred parallelization --- */
3  aOut = PE_1(aIn); bOut = PE_2(bIn); cOut = PE_3(cIn); // ...
4  /* --- inferred synchronization --- */
5  return aOut + bOut + cOut /* ... */; }
```
Control Broadcast

- **Scenario 2: Synchronization of parallel logics**
- The compiler infers parallelism from sequential code
- Insert synchronization logic to guarantee correctness

```c
#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 --- */
}
```
Control Broadcast

- Scenario 2: Synchronization of parallel logics
- The compiler infers parallelism from sequential code
- Insert synchronization logic to guarantee correctness

```c
#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 --- */
}
```
Control Broadcast

- **Scenario 2: Synchronization of parallel logics**
- The compiler infers parallelism from sequential code
- Insert synchronization logic to guarantee correctness

```c
#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 --- */
}
```
Control Broadcast

- **Scenario 2: Synchronization of parallel logics**
- The compiler infers parallelism from sequential code
- Insert synchronization logic to guarantee correctness

```c
#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 --- */
}
```

(reduce-then-broadcast)
Summary of Broadcast Types

• Data Broadcast
  • Loop unrolling: loop-invariants variables will be broadcast
  • Large buffer: logical buffer entity will become scattered memory units
  • Lead to incorrect delay prediction -> bad clock insertion

• Control Broadcast
  • Pipeline control: backpressure signals are broadcast to the whole datapath
  • Synchronization control: guarantee the correctness of concurrent execution
  • Unscalable broadcast of control signals -> not working for large designs
Outline

• Introduction
• Problem Classification
• Solution
• Experiments
Broadcast-Aware Scheduling

• Isolate the broadcast skeletons and measure the delay
Broadcast-Aware Scheduling

• Isolate the broadcast skeletons and measure the delay
• The additional delay serve as a conservative calibration

A broadcast skeleton

(a) 32 bits integers adder
(b) 32 bits FP multiplier
(c) BRAM buffer access
Broadcast-Aware Scheduling

• Example: a genome sequencing accelerator design
• Broadcast elements to 64 datapaths

```c
#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;
  }
} ....
```
Broadcast-Aware Scheduling

- Example: a genome sequencing accelerator design
- Broadcast elements to 64 datapaths

```c
#pragma HLS pipeline
#define UNROLL_FACTOR 64

for (int j = 0; j < UNROLL_FACTOR; j++) {
    // Unrolling HLS pragma
    dist_x = prev[j].x - curr.x;
    dist_y = prev[j].y - curr.y;

    if (dist_x > dist_y) dist_x = dist_y;
    if (dist_y == 0) dist_y = 1;

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

- Example: a genome sequencing accelerator design
- Broadcast elements to 64 datapaths

```c
#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;
    do = 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;
    }
} ....
```
Broadcast-Aware Scheduling

Delay of the aforementioned path
Broadcast-Aware Scheduling

Delay of the aforementioned path

Overall frequency improvements
Skid-Buffer-Based Pipeline Control

• Adopt skid buffer for flow control
Skid-Buffer-Based Pipeline Control

• Adopt skid buffer for flow control
Skid-Buffer-Based Pipeline Control

• Adopt skid buffer for flow control
Skid-Buffer-Based Pipeline Control

- Adopt skid buffer for flow control
Skid-Buffer-Based Pipeline Control

• Buffer width equals that of the pipeline output
• Different pipeline stages have different output width
Skid-Buffer-Based Pipeline Control

• Buffer width equals that of the pipeline output
• Different pipeline stages have different output width
• Dynamic programming to optimize the area overhead
Synchronization Logic Pruning

• Prune away redundant synchronization logic
Experiment Results

• > 50% improvement on our benchmarks
• For more details please check our paper :)  
• [https://github.com/Licheng-Guo/vivado-hls-broadcast-optimization](https://github.com/Licheng-Guo/vivado-hls-broadcast-optimization)

<table>
<thead>
<tr>
<th>Application</th>
<th>Broadcast type</th>
<th>Target FPGA</th>
<th>LUT (%)</th>
<th>FF (%)</th>
<th>BRAM (%)</th>
<th>DSP (%)</th>
<th>Freq (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Orig</td>
<td>Opt</td>
<td>Orig</td>
<td>Opt</td>
<td>Orig</td>
</tr>
<tr>
<td>Genome Sequencing [9]</td>
<td>Data</td>
<td>UltraScale+</td>
<td>22</td>
<td>22</td>
<td>11</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>LSTM Network [10]</td>
<td>Data</td>
<td>UltraScale+</td>
<td>8</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Face Detection [1]</td>
<td>Data</td>
<td>Kintex-7</td>
<td>21</td>
<td>22</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>Stream Buffer</td>
<td>Pipe. Ctrl. &amp; Data</td>
<td>UltraScale+</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>95</td>
</tr>
<tr>
<td>Stencil [3]</td>
<td>Pipe. Ctrl.</td>
<td>UltraScale+</td>
<td>40</td>
<td>40</td>
<td>41</td>
<td>41</td>
<td>30</td>
</tr>
<tr>
<td>Vector Arithmetic</td>
<td>Pipe. Ctrl. &amp; Sync.</td>
<td>UltraScale+</td>
<td>17</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>HBM-Based Stencil [3]</td>
<td>Pipe. Ctrl. &amp; Sync.</td>
<td>UltraScale+ HBM</td>
<td>21</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>34</td>
</tr>
<tr>
<td>Pattern Matching [4]</td>
<td>Data &amp; Sync.</td>
<td>Virtex-7</td>
<td>17</td>
<td>17</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>
Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to improve Maximum Frequency

• We classify and analyze the common types of broadcasts in HLS
• We propose methods:
  • delay model calibration to optimize the data broadcast
  • min-area skid-buffer to optimize pipeline control
  • synchronization pruning to optimize synchronization broadcast
• We bring over 50% of frequency gain to well-optimized designs.
• https://github.com/Licheng-Guo/vivado-hls-broadcast-optimization

Licheng Guo*, Jason Lau*, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang and Jason Cong
University of California Los Angeles, Cornell University