From C/C++ Code to High-Performance Dataflow Circuits

Lana Josipović

November 2022
How to perform hardware design?

High parallelism and energy efficiency
High-Level Synthesis: From Programs to Circuits

A completely new type of users for HLS!

Software application programmers

A completely new type of applications for HLS!

General-purpose code
Standard HLS

- **Create a datapath** suitable to implement the required computation
- **Create a fixed schedule at compile time** to activate the datapath components

```c
for (i=0; i<n; i++) {
    acc += x[i] * c[n-i];
}
```

![Diagram showing datapath and control flow](image)
Standard HLS

- **Create a datapath** suitable to implement the required computation
- **Create a fixed schedule at compile time** to activate the datapath components

```c
for (i=0; i<n; i++) {
    acc += x[i] * c[n-i];
}
```

![Diagram of Standard HLS](image)
Standard HLS

- **Create a datapath** suitable to implement the required computation
- Create a **fixed schedule at compile time** to activate the datapath components

```c
for (i=0; i<n; i++) {
    acc += x[i] * c[n-i];
}
```

![Diagram showing the datapath, controller, and operation schedule.](Image)
Standard HLS

• **Create a datapath** suitable to implement the required computation
• **Create a fixed schedule at compile time** to activate the datapath components

```c
for (i=0; i<n; i++) {
    acc += x[i] * c[n-i];
}
```

![Diagram of program functionality and operation schedule](image)
Standard HLS

- Create a **datapath** suitable to implement the required computation
- Create a **fixed schedule at compile time** to activate the datapath components

```c
for (i=0; i<n; i++) {
    acc += x[i] * c[n-i];
}
```

![Diagram showing the implementation of the computation with a datapath and a fixed schedule at compile time.](image)
Standard HLS

• **Create a datapath** suitable to implement the required computation
• **Create a fixed schedule at compile time** to activate the datapath components

```plaintext
for (i=0; i<n; i++) {
    acc += x[i] * c[n-i];
}
```

![Diagram of program functionality and operation schedule with static controller and 2 stages of loop iterations and clock cycles.]
Standard HLS

- **Create a datapath** suitable to implement the required computation
- **Create a fixed schedule at compile time** to activate the datapath components

```c
for (i=0; i<n; i++) {
  acc += x[i] * c[n-i];
}
```

Low throughput: slow execution
Standard HLS

- **Create a datapath** suitable to implement the required computation
- **Create a fixed schedule at compile time** to activate the datapath components

```plaintext
for (i=0; i<n; i++) {
    acc += x[i] * c[n-i];
}
```

- **Naïve schedule:**
- **Pipelined schedule:**

**High throughput: fast execution**
The Limitations of Static Scheduling

for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}

- Static scheduling (standard HLS tool)
  - Inferior when memory accesses cannot be disambiguated at compile time

- Dynamic scheduling
  - Maximum parallelism: Only serialize memory accesses on actual dependencies

RAW dependency
A Different Way to Do HLS

Static scheduling (standard HLS tool): decide at compile time when each operation executes

Dynamic scheduling (our HLS approach): decide at runtime when each operation executes
A Different Way to Do HLS

**Static scheduling (standard HLS tool):** decide at compile time when each operation executes

**Dynamic scheduling (our HLS approach):** decide at runtime when each operation executes
A Different Way to Do HLS

Static scheduling (standard HLS tool): decide at compile time when each operation executes

Dynamic scheduling (our HLS approach): decide at runtime when each operation executes
Dataflow Circuits

• **Asynchronous circuits**: operators triggered when inputs are available
  – Budiu et al. Dataflow: A complement to superscalar. ISPASS’05.

• **Dataflow, latency-insensitive, elastic**: the *synchronous* version of it
  – Cortadella et al. Synthesis of synchronous elastic architectures. DAC’06.
  – Carloni et al. Theory of latency-insensitive design. TCAD’01.
  – Jacobson et al. Synchronous interlocked pipelines. ASYNC’02.

High-level synthesis of dataflow circuits
HLS of Dynamically Scheduled Circuits
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining

Resource sharing

Mul 1
Mul 2
Mul 1/2
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining

Resource sharing

Mul 1
Mul 2
Mul 1/2

Reaping the benefits of dynamic scheduling

Out-of-order memory

Speculative execution
Dataflow Circuits

- We use the **SELF (Synchronous ELastic Flow)** protocol
  - Cortadella et al. Synthesis of synchronous elastic architectures. DAC’06.
- Every component communicates via a pair of handshake signals
- **Make scheduling decisions at runtime**
  - As soon as all conditions for execution are satisfied, an operation starts
Dataflow Circuits

• We use the **SELF (Synchronous ELastic Flow)** protocol
  – Cortadella et al. Synthesis of synchronous elastic architectures. DAC’06.
• Every component communicates via a pair of handshake signals
• **Make scheduling decisions at runtime**
  – As soon as all conditions for execution are satisfied, an operation starts

![Dataflow Circuit Diagram](image-url)
Dataflow Circuits

• We use the **SELF (Synchronous ELastic Flow)** protocol
  – Cortadella et al. Synthesis of synchronous elastic architectures. DAC’06.

• Every component communicates via a pair of handshake signals

• **Make scheduling decisions at runtime**
  – As soon as all conditions for execution are satisfied, an operation starts
Dataflow Circuits

• We use the **SELF (Synchronous ELastic Flow)** protocol
  – Cortadella et al. Synthesis of synchronous elastic architectures. DAC’06.
• Every component communicates via a pair of handshake signals
• **Make scheduling decisions at runtime**
  – As soon as all conditions for execution are satisfied, an operation starts

![Dataflow Circuit Diagram]

Component 1 ➔ valid ➔ ready ➔ data ➔ Component 2

---

Circuit clock

<table>
<thead>
<tr>
<th>Valid</th>
<th>Ready</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>D1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D3</td>
</tr>
</tbody>
</table>
Dataflow Components

- Fork
- Join
- Branch
- Merge
Dataflow Components
Dataflow Components

- Fork
- Join
- Branch
- Merge
Dataflow Components
Dataflow Components

- Fork
- Join
- Branch
- Merge
Dataflow Components

- Fork
- Join
- Branch
- Merge
From Program to Dataflow Circuit

for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}

Josipović, Ghosal, and Ienne. Dynamically Scheduled High-Level Synthesis. FPGA 2018 **Best Paper Award Nominee**

Josipović, Brisk, and Ienne. From C to Elastic Circuits. Asilomar 2017
From Program to Dataflow Circuit

for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}

Josipović, Ghosal, and Ienne. Dynamically Scheduled High-Level Synthesis. FPGA 2018 Best Paper Award Nominee
Josipović, Brisk, and Ienne. From C to Elastic Circuits. Asilomar 2017
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
From Program to Dataflow Circuit

Single token on cycle, in-order tokens in noncyclic paths

Josipović, Ghosal, and Ienne. Dynamically Scheduled High-Level Synthesis. FPGA 2018 Best Paper Award Nominee
Josipović, Brisk, and Ienne. From C to Elastic Circuits. Asilomar 2017
Backpressure from slow paths prevents pipelining
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining

Resource sharing

Mul 1, Mul 2, Mul 1/2

Reaping the benefits of
dynamic scheduling

Out-of-order memory

Speculative execution
Performance Optimizations

• **Static HLS**: modulo scheduling
  – Cong et al. An efficient and versatile scheduling algorithm based on SDC formulation. DAC’06.
Performance Optimizations

- **Static HLS:** modulo scheduling
  - Cong et al. An efficient and versatile scheduling algorithm based on SDC formulation. DAC’06.

- **Asynchronous dataflow:** slack matching
  - Venkataramani et al. Leveraging protocol knowledge in slack matching. ICCAD’06.
Performance Optimizations

• **Static HLS**: modulo scheduling
  – Cong et al. An efficient and versatile scheduling algorithm based on SDC formulation. DAC’06.

• **Asynchronous dataflow**: slack matching
  – Venkataramani et al. Leveraging protocol knowledge in slack matching. ICCAD’06.

Optimize throughput and clock period of dataflow circuits obtained from high-level code
Buffers in Dataflow Circuits

• Buffer capacity
  – Number of data slots
  – Use buffers as FIFOs to accumulate data
Buffers in Dataflow Circuits

• **Buffer capacity**
  – Number of data slots
  – Use buffers as FIFOs to accumulate data

• **Buffer transparency**
  – Indicates whether a buffer has a combinational path between input and output
  – Nontransparent (N-buff) and transparent (T-buff)
Buffers in Dataflow Circuits

• **Buffer capacity**
  – Number of data slots
  – Use buffers as FIFOs to accumulate data

• **Buffer transparency**
  – Indicates whether a buffer has a combinational path between input and output
  – Nontransparent (N-buff) and transparent (T-buff)
Buffers in Dataflow Circuits

- **Buffer capacity**
  - Number of data slots
  - Use buffers as FIFOs to accumulate data

- **Buffer transparency**
  - Indicates whether a buffer has a combinational path between input and output
  - Nontransparent (N-buff) and transparent (T-buff)

- **Circuit functionality**
  - Buffer insertion does not affect circuit functionality

---

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Buffers in Dataflow Circuits

- **Buffer capacity**
  - Number of data slots
  - Use buffers as FIFOs to accumulate data

- **Buffer transparency**
  - Indicates whether a buffer has a combinational path between input and output
  - Nontransparent (N-buff) and transparent (T-buff)

- **Circuit functionality**
  - Buffer insertion does not affect circuit functionality
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}

Nontransparent buffer breaks the combinational loop

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Inserting Buffers

for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}

Transparent buffers of larger capacity regulate throughput

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Inserting Buffers

for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
Inserting Buffers

**NOW** (with buffers)

```
Start: i=0

Merge

Fork

LD x[i]

M

Merge

Buff

Fork

LD weight[i]

ST hist[x[i]]

Exit: i=N
```

4 stages comb.

**BEFORE** (without buffers)

```
Start: i=0

Merge

Fork

LD x[i]

M

Merge

Fork

LD weight[i]

ST hist[x[i]]

Exit: i=N
```

4 stages comb.

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Marked Graphs

• **Choice-free** Petri nets: concurrent behavior, no conditional execution
• Timing optimizations are known
  – Bufistov et al. A general model for performance optimization of sequential systems, ICCAD’07.

Josipović, Sheikhha, Guerrieri, Lenne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Marked Graphs

- **Choice-free** Petri nets: concurrent behavior, no conditional execution
- Timing optimizations are known
  - Bufistov et al. A general model for performance optimization of sequential systems, ICCAD’07.
Marked Graphs

- **Choice-free** Petri nets: concurrent behavior, no conditional execution
- Timing optimizations are known
  - Bufistov et al. A general model for performance optimization of sequential systems, ICCAD’07.

Dataflow circuits obtained from high-level code feature choices

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Optimizing Performance

1. Extract choice-free dataflow circuits from the dataflow circuit
2. Determine buffer placement and properties (capacity, transparency)
   - Satisfy the required clock period
   - Maximize the throughput of the choice-free circuits

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Optimizing Performance

1. Extract choice-free dataflow circuits from the dataflow circuit
   - ILP model to iteratively extract control flow graph cycles based on their execution frequency (obtained through profiling)
   - Identify dataflow components which belong to each extracted control flow cycle

for (i = 0; i < N; i++)
for (j = 0; j < M; j++)
...

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Optimizing Performance

1. Extract choice-free dataflow circuits from the dataflow circuit
   - ILP model to iteratively extract control flow graph cycles based on their execution frequency (obtained through profiling)
   - Identify dataflow components which belong to each extracted control flow cycle

```plaintext
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        ...
```

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Optimizing Performance

2. Determine buffer placement and properties (capacity, transparency)
   - **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   - Single choice-free subgraph:

![Diagram of choice-free subgraph with buffers and delays]

Target CP = 4 ns

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 *Best paper award*
Optimizing Performance

2. Determine buffer placement and properties (capacity, transparency)
   — MILP model to optimize throughput of the choice-free circuits under a clock period constraint
   — Single choice-free subgraph:

   Objective: maximize throughput for a target period and minimize buffer slot count

   \[
   \text{max: } \Phi - \lambda \cdot \sum N_c \quad \text{throughput small const. buffer slots}
   \]

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Optimizing Performance

2. Determine buffer placement and properties (capacity, transparency)
   - **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   - Single choice-free subgraph:

   **Objective:** maximize throughput for a target period and minimize buffer slot count

   **Path constraints:** add buffers to meet target clock period

   \[
   \text{target period} \quad N \text{-buff} \\
   t_{\text{out}}^c \geq t_{\text{in}}^c - CP \cdot R_c \\
   CP \geq t_{\text{in}}^c \geq t_{\text{out}}^c + D_u
   \]

   in/out arrival time \quad \text{unit comb. delay}

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Optimizing Performance

2. Determine buffer placement and properties (capacity, transparency)
   - **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   - Single choice-free subgraph:

   **Objective:** maximize throughput for a target period and minimize buffer slot count

   **Path constraints:** add buffers to meet target clock period

   \[
   \begin{align*}
   \text{max: } & \Phi - \lambda \cdot \sum_c N_c \\
   \text{target period: } & N\text{-buff} \\
   t_{c_{\text{out}}}^{\text{out}} & \geq t_{c_{\text{in}}}^{\text{in}} - \text{CP} \cdot R_c \\
   \text{CP} & \geq t_{c_{2}}^{\text{in}} \geq t_{c_{1}}^{\text{out}} + D_u \\
   \end{align*}
   \]

   *in/out arrival time, unit comb. delay*

   **Target CP = 4 ns**
Optimizing Performance

2. Determine buffer placement and properties (capacity, transparency)
   — **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   — Single choice-free subgraph:

   **Objective:** maximize throughput for a target period and minimize buffer slot count

   **Path constraints:** add buffers to meet target clock period

   **Throughput constraints:** compute average number of tokens in a channel

   \[
   \begin{align*}
   \text{max:} & \quad \Phi - \lambda \cdot \sum C \n \end{align*}
   \]

   \[
   \begin{align*}
   t_c^{\text{out}} & \geq t_c^{\text{in}} - CP \cdot R_c \\
   CP & \geq t_c^{\text{in}} \geq t_c^{\text{out}} + D_u \\
   \theta_c & = B_b + r_v - r_u \\
   \theta_c & \geq \Phi + R_c - 1
   \end{align*}
   \]

   Target CP = 4 ns

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
2. Determine buffer placement and properties (capacity, transparency)
   — **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   — Single choice-free subgraph:

   **Objective:** maximize throughput for a target period and minimize buffer slot count

   **Path constraints:** add buffers to meet target clock period

   **Throughput constraints:** compute average number of tokens in a channel

   \[
   \text{max: } \Phi - \lambda \cdot \sum C_P \\
   t_{c_{\text{out}}} \geq t_{c_{\text{in}}} - CP \cdot R_c \\
   CP \geq t_{c_{\text{in}}} \geq t_{c_{\text{out}}} + D_u \\
   \theta_c = B_b + r_v - r_u \\
   \theta_c \geq \Phi + R_c - 1
   \]

   Target CP = 4 ns
   Throughput: \( \Phi = 1 \)
Optimizing Performance

2. Determine buffer placement and properties (capacity, transparency)
   - **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   - Single choice-free subgraph:

   **Objective:** maximize throughput for a target period and minimize buffer slot count

   **Path constraints:** add buffers to meet target clock period

   **Throughput constraints:** compute average number of tokens in a channel

   \[ \max: \Phi - \lambda \cdot \sum_c N_c \]

   \[ t_{c_{out}} \geq t_{c_{in}} - CP \cdot R_c \]

   \[ CP \geq t_{c_{out}}^{in} \geq t_{c_{out}}^{out} + D_u \]

   \[ \theta_c = B_b + r_v - r_u \]

   \[ \theta_c \geq \Phi + R_c - 1 \]

   \[ \text{channel occupancy} \]

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Optimizing Performance

2. Determine buffer placement and properties (capacity, transparency)
   — MILP model to optimize throughput of the choice-free circuits under a clock period constraint
   — Single choice-free subgraph:

   **Objective:** maximize throughput for a target period and minimize buffer slot count

   **Path constraints:** add buffers to meet target clock period

   **Throughput constraints:** compute average number of tokens in a channel

   **Buffer sizing:** add buffer slots to avoid backpressure and maximize throughput

   \[
   \begin{align*}
   \max: \Phi - \lambda \cdot \sum \limits_c N_c \\
   t_{c, out}^{in} \geq t_{c, in}^{out} - CP \cdot R_c \\
   CP \geq t_{c, 2}^{in} \geq t_{c, 1}^{out} + D_u \\
   \Theta_c = B_b + r_v - r_u \\
   \Theta_c \geq \Phi + R_c - 1 \\
   \end{align*}
   \]

   \[
   \begin{align*}
   \text{channel emptiness} & \quad N_c \geq \Theta_c + \Theta_c^p \\
   \text{buffer slots} & \quad \text{Target CP = 4 ns} \\
   \text{Throughput: } \Phi = 1
   \end{align*}
   \]
Optimizing Performance

2. Determine buffer placement and properties (capacity, transparency)
   
   - **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   
   - Single choice-free subgraph:

   ![Diagram of a graph with nodes labeled Fork, Store, and the constraints 
   \( \Theta_c = 1 \), \( N_c = 2 \), and \( \Theta_c = 1 \).]

   **Objective:** maximize throughput for a target period and minimize buffer slot count

   **Path constraints:** add buffers to meet target clock period

   **Throughput constraints:** compute average number of tokens in a channel

   **Buffer sizing:** add buffer slots to avoid backpressure and maximize throughput

   
   \[
   \begin{align*}
   \max : & \quad \Phi - \lambda \cdot \sum_c N_c \\
   t_c^{\text{out}} & \geq t_c^{\text{in}} - CP \cdot R_c \\
   CP & \geq t_c^{\text{in}} \geq t_c^{\text{out}} + D_u \\
   \Theta_c & = B_b + r_v - r_u \\
   \Theta_c & \geq \Phi + R_c - 1
   \end{align*}
   \]

   \[N_c \geq \Theta_c + \Theta_c^*\]

   **channel emptiness**

   **buffer slots**

Josipović, Sheikhha, Guerrieri, Lenne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
2. Determine buffer placement and properties (capacity, transparency)

   — **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   — Single choice-free subgraph:

   **Objective:** maximize throughput for a target period and minimize buffer slot count

   **Path constraints:** add buffers to meet target clock period

   **Throughput constraints:** compute average number of tokens in a channel

   **Buffer sizing:** add buffer slots to avoid backpressure and maximize throughput

   ![Diagram of buffer placement and properties](image)

   Target CP = 3 ns

   \[
   \begin{align*}
   \text{max: } \Phi - \lambda \cdot \sum_c N_c \\
   t_{c_{\text{out}}} &\geq t_{c_{\text{in}}} - CP \cdot R_c \\
   CP &\geq t_{c_{1,\text{in}}} \geq t_{c_{1,\text{out}}} + D_u \\
   \Theta_c &= B_b + r_v - r_u \\
   \Theta_c &\geq \Phi + R_c - 1 \\
   N_c &\geq \Theta_c + \Theta_c^°
   \end{align*}
   \]
Optimizing Performance

2. Determine buffer placement and properties (capacity, transparency)
   — **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   — Single choice-free subgraph:

   ![](image)

   **Target CP = 3 ns**

   **Objective:** maximize throughput for a target period and minimize buffer slot count

   
   

   **Path constraints:** add buffers to meet target clock period

   

   **Throughput constraints:** compute average number of tokens in a channel

   

   **Buffer sizing:** add buffer slots to avoid backpressure and maximize throughput

   

   \[
   \text{max: } \Phi - \lambda \cdot \sum_c N_c
   \]

   \[
   t_c^{\text{out}} \geq t_c^{\text{in}} - CP \cdot R_c
   \]

   \[
   CP \geq t_{c_2}^{\text{in}} \geq t_{c_1}^{\text{out}} + D_u
   \]

   \[
   \Theta_c = B_b + r_v - r_u
   \]

   \[
   \Theta_c \geq \Phi + R_c - 1
   \]

   \[
   N_c \geq \Theta_c + \Theta_c^\circ
   \]
Optimizing Performance

2. Determine buffer placement and properties (capacity, transparency)
   - **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   - Single choice-free subgraph:

   Objective: maximize throughput for a target period and minimize buffer slot count

   Path constraints: add buffers to meet target clock period

   Throughput constraints: compute average number of tokens in a channel

   Buffer sizing: add buffer slots to avoid backpressure and maximize throughput

   \[
   \text{max: } \Phi - \lambda \cdot \sum_c N_c
   \]

   \[
   t_{c,2}^{\text{out}} \geq t_{c,1}^{\text{in}} - CP \cdot R_c
   \]

   \[
   CP \geq t_{c,2}^{\text{in}} \geq t_{c,1}^{\text{out}} + D_u
   \]

   \[
   \theta_c = B_b + r_v - r_u
   \]

   \[
   \theta_c \geq \Phi + R_c - 1
   \]

   \[
   N_c \geq \theta_c + \theta_c^\circ
   \]

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Optimizing Performance

2. Determine buffer placement and properties (capacity, transparency)
   — **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   — Single choice-free subgraph:

   **Objective:** maximize throughput for a target period and minimize buffer slot count

   **Path constraints:** add buffers to meet target clock period

   **Throughput constraints:** compute average number of tokens in a channel

   **Buffer sizing:** add buffer slots to avoid backpressure and maximize throughput

   \[
   \max: \Phi - \lambda \cdot \sum_c N_c \\
   t_{c_{\text{out}}} \geq t_{c_{\text{in}}} - CP \cdot R_c \\
   CP \geq t_{c_{\text{in}}}^{t_{c_{\text{in}}}^{\text{out}}} + D_u \\
   \theta_c = B_b + r_v - r_u \\
   \theta_c \geq \Phi + R_c - 1 \\
   N_c \geq \theta_c + \theta_c^\circ
   \]

   Target CP = 3 ns
   Throughput: \( \Phi = 1/2 \)
Optimizing Performance

2. Determine buffer placement and properties (capacity, transparency)
   — **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   — Single choice-free subgraph:

   **Objective:** maximize throughput for a target period and minimize buffer slot count

   \[
   \max: \Phi - \lambda \cdot \sum_c N_c
   \]

   **Path constraints:** add buffers to meet target clock period

   \[
   t_{c_2}^{\text{out}} \geq t_{c_1}^{\text{in}} - CP \cdot R_c
   \]

   \[
   CP \geq t_{c_1}^{\text{in}} \geq t_{c_1}^{\text{out}} + D_u
   \]

   **Throughput constraints:** compute average number of tokens in a channel

   \[
   \Theta_c = B_b + r_v - r_u
   \]

   \[
   \Theta_c \geq \Phi + R_c - 1
   \]

   **Buffer sizing:** add buffer slots to avoid backpressure and maximize throughput

   \[
   N_c \geq \Theta_c + \Theta_c^\circ
   \]

For entire dataflow graph

For choice-free subgraph

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 **Best paper award**
2. Determine buffer placement and properties (capacity, transparency)
   
   — **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   
   — **Multiple** choice-free subgraphs:

   **Objective:** \( \text{max. weighted throughput sum} \) for a target period and minimize buffer slot count

   \[
   \max \sum_i w_i \cdot \Phi_i - \lambda \sum_c N_c
   \]

   **Path constraints:** add buffers to meet target clock period

   \[
   t_c^{\text{out}} \geq t_c^{\text{in}} - CP \cdot R_c \\
   CP \geq t_c^{\text{in}} \geq t_c^{\text{out}} + D_u
   \]

   **Throughput constraints:** compute average number of tokens in a channel

   \[
   \Theta_c = B_b + r_v - r_u \\
   \Theta_c \geq \Phi + R_c - 1
   \]

   **Buffer sizing:** add buffer slots to avoid backpressure and maximize throughput

   \[
   N_c \geq \Theta_c + \Theta_c^\circ
   \]

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Optimizing Performance

2. Determine buffer placement and properties (capacity, transparency)
   — **MILP model** to optimize throughput of the choice-free circuits under a clock period constraint
   — **Multiple** choice-free subgraphs:

   **Objective:** max. weighted throughput sum for a target period and minimize buffer slot count

   \[
   \max \sum_i w_i \cdot \Phi_i - \lambda \cdot \sum_c N_c
   \]

   **Path constraints:** add buffers to meet target clock period

   \[
   t_c^{out} \geq t_c^{in} - CP \cdot R_c
   \]

   \[
   CP \geq t_c^{in} \geq t_c^{out} + D_u
   \]

   **Throughput constraints:** compute average number of tokens in a channel

   \[
   \Theta_c = B_b + r_v - r_u
   \]

   \[
   \Theta_c \geq \Phi + R_c - 1
   \]

   **Buffer sizing:** add buffer slots to avoid backpressure and maximize throughput

   \[
   N_c \geq \Theta_c + \Theta_c^c
   \]

   **Optimal buffer placement & sizing for the given execution profile**
Achieving High Frequency

- Distributed control mechanism → long combinational paths and frequency degradation

Rizzi, Guerrieri, Ienne, and Josipović. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. FPL 2022
Achieving High Frequency

- Distributed control mechanism $\rightarrow$ long combinational paths and **frequency degradation**

Critical path spans through data and control network

Rizzi, Guerrieri, Ienne, and Josipović. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. FPL 2022
Achieving High-Frequency

• Distributed control mechanism  $\rightarrow$ long combinational paths and frequency degradation

Rizzi, Guerrieri, Ienne, and Josipović. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. FPL 2022
Achieving High-Frequency

- Distributed control mechanism $\rightarrow$ long combinational paths and frequency degradation

Naïve timing model

Our timing model

Accurate model of all combinational delays and interconnects between the data and the control signals

Rizzi, Guerrieri, Ienne, and Josipović. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. FPL 2022
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining

Fork
Load
ready
FIFO
stall
Store

Resource sharing

Mul 1
Mul 2
Mul 1/2

Reaping the benefits of dynamic scheduling

Out-of-order memory

Speculative execution

Load
LD
ST
Memory

Store

Save
Fork

Speculator
Commit

Save

Commit
Saving Resources through Sharing

- Static HLS: share units between operations which execute in **different clock cycles**
- Dynamic HLS: share units based on their **average utilization** with tokens

```c
for (i = 0; i < N; i++) {
    a[i] = a[i]*x;
    b[i] = b[i]*y;
}
```

Josipović, Marmet, Guerrieri, and Ienne. Resource Sharing in Dataflow Circuits. FCCM 2022. **Best Paper Award Nominee**
Saving Resources through Sharing

- Static HLS: share units between operations which execute in different clock cycles
- Dynamic HLS: share units based on their average utilization with tokens

for (i = 0; i < N; i++) {
    a[i] = a[i]*x;
    b[i] = b[i]*y;
}

Units fully utilized (high throughput)

Sharing not possible without damaging throughput

Use MILP (performance optimization) information to decide what to share

Saving Resources through Sharing

- Static HLS: share units between operations which execute in **different clock cycles**
- Dynamic HLS: share units based on their **average utilization** with tokens

```c
for (i = 0; i < N; i++) {
    a[i] = a[i]*x;
    b[i] = b[i]*y;
}
```

Sharing possible without damaging throughput

Units underutilized (low throughput)

Use MILP (performance optimization) information to decide what to share

Saving Resources through Sharing

- Static HLS: share units between operations which execute in different clock cycles
- Dynamic HLS: share units based on their average utilization with tokens

for (i = 0; i < N; i++) {
    a[i] = a[i]*x;
    b[i] = b[i]*y;
}

Sharing mechanism for deadlock-free execution

for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
Inserting Buffers

for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}

Buffers for high throughput
Inserting Buffers

for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}

1: x[0]=5 → ld hist[5]; st hist[5];
2: x[1]=4 → ld hist[4]; st hist[4];
3: x[2]=4 → ld hist[4]; st hist[4];

RAW dependency not honored!

What about memory?
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

- Pipelining
  - Fork
  - Load
  - FIFO
  - Ready
  - Stall
  - Store

- Resource sharing
  - Mul 1
  - Mul 2
  - Mul 1/2

Reaping the benefits of dynamic scheduling

Out-of-order memory

Speculative execution

BB start

Memory

Load

LD

ST

Store

Save

Speculator

Fork

Commit
We Need a Load-Store Queue (LSQ)!

- Traditional processor LSQs allocate memory instructions **in program order**

- Dataflow circuits have **no notion of program order**

 annual processor LSQs allocate memory instructions **in program order**

loop: lw $t2, 0(t4)  
    lw $t3, 100(t4)  
mul $t5, $t2, $t3  
addi $t5, $t5, $t1  
sw $t5, 100(t4)  
addi $t1, $t1, 4  
bne $t6, $t1, loop

How to supply program order to the LSQ?
LSQ Allocation

- An LSQ for dataflow circuits whose only difference is in the allocation policy:
  - Static knowledge of memory access program order inside each basic block
  - Dynamic knowledge of the sequence of basic blocks from the dataflow circuit

```c
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
```

LSQ Allocation

• An LSQ for dataflow circuits whose only difference is in the allocation policy:
  – **Static knowledge** of memory access program order inside each basic block
  – **Dynamic knowledge** of the sequence of basic blocks from the dataflow circuit

```
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
```
LSQ Allocation

- An LSQ for dataflow circuits whose only difference is in the allocation policy:
  - **Static knowledge** of memory access program order inside each basic block
  - **Dynamic knowledge** of the sequence of basic blocks from the dataflow circuit

```c
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
```
LSQ Allocation

• An LSQ for dataflow circuits whose only difference is in the allocation policy:
  – Static knowledge of memory access program order inside each basic block
  – Dynamic knowledge of the sequence of basic blocks from the dataflow circuit

```java
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
```

LSQ Allocation

- An LSQ for dataflow circuits whose only difference is in the *allocation policy*:
  - **Static knowledge** of memory access program order inside each basic block
  - **Dynamic knowledge** of the sequence of basic blocks *from the dataflow circuit*

```c
for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}
```

Josipović, Brisk, and Ienne. An Out-of-Order Load-Store Queue for Spatial Computing. CASES 2017 *Best Paper Award Nominee*

Dataflow Circuit with the LSQ

for (i=0; i<N; i++) {
    hist[x[i]] = hist[x[i]] + weight[i];
}

1: x[0]=5 → ld hist[5]; st hist[5];
2: x[1]=4 → ld hist[4]; st hist[4];
3: x[2]=4 → ld hist[4]; st hist[4];

High-throughput pipeline with memory dependencies honored
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining

Resource sharing

Reaping the benefits of dynamic scheduling

Out-of-order memory

Speculative execution

Mul 1

Mul 2

Mul 1/2
Speculation for Extracting Parallelism

• **Static HLS approaches:** predication, partial load-store speculation
Speculation for Extracting Parallelism

- **Static HLS approaches:** predication, partial load-store speculation

- **Dataflow circuits:** lenient execution, early evaluation
  - Budiu et al. Dataflow: A complement to superscalar. ISPASS ’05.
  - Cortadella et al. Synchronous elastic circuits with early evaluation and token counterflow. DAC '07.

- **Dataflow-oriented processors:** squash and replay mechanism
  - Desikan et al. Scalable selective re-execution for EDGE architectures. ASPLOS’04.
Speculation for Extracting Parallelism

- **Static HLS approaches**: predication, partial load-store speculation

- **Dataflow circuits**: lenient execution, early evaluation
  - Budiu et al. Dataflow: A complement to superscalar. ISPASS ’05.
  - Cortadella et al. Synchronous elastic circuits with early evaluation and token counterflow. DAC '07.

- **Dataflow-oriented processors**: squash and replay mechanism
  - Desikan et al. Scalable selective re-execution for EDGE architectures. ASPLOS'04.

How to enable generic forms of speculation in dataflow circuits?
float d=0.0; x=100.0; int i=0;

do {
    d = a[i] + b[i];
    i++;
} while (d<x);
float d=0.0; x=100.0; int i=0;

```c
    do {
        d = a[i] + b[i];
        i++;
    } while (d<x);
```

Nonspeculative Dataflow Circuit
float d=0.0; x=100.0; int i=0;

do {
    d = a[i] + b[i];
    i++;
} while (d<x);
Nonspeculative Dataflow Circuit

```plaintext
float d=0.0; x=100.0; int i=0;

do {
    d = a[i] + b[i];
    i++;}
while (d<x);
```
Nonspeculative vs. Speculative System

Long control flow decision prevents pipelining
Nonspeculative vs. Speculative System
Speculation in Dataflow Circuits

• Contain speculation in a region of the circuit delimited by special components
  – Issue speculative tokens (pieces of data which might or might not be correct)
  – Squash and replay in case of misspeculation
Speculation in Dataflow Circuits

- Contain speculation in a region of the circuit delimited by special components
  - Issue speculative tokens (pieces of data which might or might not be correct)
  - Squash and replay in case of misspeculation
Speculation in Dataflow Circuits

- Contain speculation in a region of the circuit delimited by special components
  - Issue speculative tokens (pieces of data which might or might not be correct)
  - Squash and replay in case of misspeculation

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Speculation in Dataflow Circuits

- Contain speculation in a region of the circuit delimited by special components
  - Issue speculative tokens (pieces of data which might or might not be correct)
  - Squash and replay in case of misspeculation
Speculation in Dataflow Circuits

- Contain speculation in a region of the circuit delimited by special components
  - Issue speculative tokens (pieces of data which might or might not be correct)
  - Squash and replay in case of misspeculation

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Components for Speculation

• Speculator
  – Dataflow component which can, besides its standard functionality, also inject tokens before receiving any at its input(s)
  – Branch Speculator, LSQ
Components for Speculation

- Speculator
  - Dataflow component which can, besides its standard functionality, also inject tokens before receiving any at its input(s)
  - Branch Speculator, LSQ
Components for Speculation

- Save units
  - Input boundary of the speculative region
  - Reissues when previous computation is squashed

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Components for Speculation

• Save units
  – Input boundary of the speculative region
  – Reissues when previous computation is squashed

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Components for Speculation

• Commit units
  – Output boundary of the speculative region
  – Propagate speculative tokens that turn out to be correct, squash misspeculated data

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Components for Speculation

• Commit units
  – Output boundary of the speculative region
  – Propagate speculative tokens that turn out to be correct, squash misspeculated data

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Placing the Components for Speculation

• Save units
  – On each path to any component that could combine the token with a speculative
  – As close as possible to the paths carrying speculative tokens
Placing the Components for Speculation

• Save units
  – On each path to any component that could combine the token with a speculative
  – As close as possible to the paths carrying speculative tokens

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Placing the Components for Speculation

- Commit units
  - On each path from the Speculator to an exit point, a store unit, or the Speculator
  - As far as possible from the Speculator
Placing the Components for Speculation

- Commit units
  - On each path from the Speculator to an exit point, a store unit, or the Speculator
  - As far as possible from the Speculator
Speculative Dataflow Circuit

Start, i=0

Merge

Buff

Fork

Load a[i]  Load b[i]

Branch

End

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Speculator instead of regular Branch
Speculative Dataflow Circuit

Start, i=0

Merge

Buff

Fork

Load a[i]

Load b[i]

Save

Spec. Branch

Input boundary: Save units

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Speculative Dataflow Circuit

Start, i=0

Merge

Buff

Fork

Load a[i]

Load b[i]

Output boundary: Commit units

Speculative Dataflow Circuit by Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Speculative Dataflow Circuit

Start, i=0

Merge

Buff

Fork

Load a[i]
Load b[i]

Commit

Save

Spec. Branch

Commit

End

Output boundary: Commit units

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Speculative Dataflow Circuit

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Speculative Dataflow Circuit

Single speculation at a time cannot achieve maximum parallelism

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Increasing Performance

• Merging the Save and Commit unit on cyclic paths

  **Commit unit:**
  • Stalls speculative tokens
  • Discards mis-speculated tokens

  ![Commit unit diagram]

  **Save unit:**
  • Saves and reissues tokens

  ![Save unit diagram]

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Increasing Performance

- Merging the Save and Commit unit on cyclic paths

**Commit unit:**
- Stalls speculative tokens
- Discards misspeculated tokens

**Save unit:**
- Saves and reissues tokens

**Save-Commit unit:**
- Lets speculative tokens pass
- Discards misspeculated tokens
- Saves and reissues tokens

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Increasing Performance

- Merging the Save and Commit unit on cyclic paths

**Commit unit:**
- Stalls speculative tokens
- Discards misspeculated tokens

**Save unit:**
- Saves and reissues tokens

**Save-Commit unit:**
- Lets speculative tokens pass
- Discards misspeculated tokens
- Saves and reissues tokens

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Increasing Performance

- Merging the Save and Commit unit on cyclic paths

**Commit unit:**
- Stalls speculative tokens
- Discards misspeculated tokens

**Save unit:**
- Saves and reissues tokens

**Save-Commit unit:**
- Lets speculative tokens pass
- Discards misspeculated tokens
- Saves and reissues tokens

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Increasing Performance

- Merging the Save and Commit unit on cyclic paths

**Commit unit:**
- Stalls speculative tokens
- Discards misspeculated tokens

**Save unit:**
- Saves and reissues tokens

**Save-Commit unit:**
- Lets speculative tokens pass
- Discards misspecified tokens
- Saves and reissues tokens

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Increasing Performance

• Merging the Save and Commit unit on cyclic paths

Commit unit:
  • Stalls speculative tokens
  • Discards misspeculated tokens

Save unit:
  • Saves and reissues tokens

Save-Commit unit:
  • Lets speculative tokens pass
  • Discards misspeculated tokens
  • Saves and reissues tokens

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
Speculative Dataflow Circuit

Start, i=0

Merge

Buff

Fork

Load a[i]

Load b[i]

+ +

d x

Stalls speculative tokens

Saves a single nonspeculative token

Commit

Save

Spec. Branch

Commit

End

Single speculation at a time cannot achieve maximum parallelism
Speculative Dataflow Circuit

Stalls speculative tokens
Saves a single nonspeculative token

Start, i=0

Merge
Fork
Load a[i]
Load b[i]

Commit
Save
Spec. Branch
Commit
End

Start, i=0

Merge
Fork
Load a[i]
Load b[i]

Save-Commit
Spec. Branch
Commit
End

Let's speculative tokens pass + saves multiple tokens
Speculative Dataflow Circuit

High-throughput speculative pipeline

Josipović, Guerrieri, and Ienne. Speculative Dataflow Circuits. FPGA 2019
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining

Fork
Load
FIFO
ready
stall
Store

Resource sharing

Mul 1
Mul 2
Mul 1/2

Reaping the benefits of dynamic scheduling

Out-of-order memory

Load
LD
ST
Store
Memory

Speculative execution

Save
Speculator

Save
Fork
Commit

Static HLS vs. dynamic HLS?
Dynamatic: An Open-Source HLS Compiler

- From C/C++ to synthesizable dataflow circuit description
Experimental Results

- Resource utilization and execution time of the dataflow designs, normalized to the corresponding static designs produced by Vivado HLS.
Experimental Results

- Resource utilization and execution time of the dataflow designs, normalized to the corresponding static designs produced by Vivado HLS.
Experimental Results

- Resource utilization and execution time of the dataflow designs, **normalized to the corresponding static designs** produced by Vivado HLS

![Graph showing resource utilization and execution time comparison](image-url)

Josipović, Guerrieri, and Ienne. Synthesizing General-Purpose Code into Dynamically Scheduled Circuits. CASM 2021
Experimental Results

• Resource utilization and execution time of the dataflow designs, normalized to the corresponding static designs produced by Vivado HLS

Reduced execution time in irregular benchmarks (speedup of up to 14.9X)

Josipović, Guerrieri, and Ienne. Synthesizing General-Purpose Code into Dynamically Scheduled Circuits. CASM 2021
Experimental Results

- Resource utilization and execution time of the dataflow designs, normalized to the corresponding static designs produced by Vivado HLS

Reduced execution time in irregular benchmarks (speedup of up to 14.9X)
Experimental Results

• Resource utilization and execution time of the dataflow designs, normalized to the corresponding static designs produced by Vivado HLS

LSQ causes significant resource overheads
Experimental Results

- Resource utilization and execution time of the dataflow designs, normalized to the corresponding static designs produced by Vivado HLS.

LSQ causes significant resource overheads.

Experimental Results

• Resource utilization and execution time of the dataflow designs, normalized to the corresponding static designs produced by Vivado HLS

Static & dynamic HLS have the same pipelining capabilities

Josipović, Guerrieri, and Ienne. Synthesizing General-Purpose Code into Dynamically Scheduled Circuits. CASM 2021
Static vs. Dynamic Scheduling

- **Statically Scheduled** → “Compiler does the job”
- **Dynamically Scheduled** → “Hardware does the job”

<table>
<thead>
<tr>
<th>VLIW Processors</th>
<th>Out-of-Order Superscalar Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional HLS</td>
<td>Dataflow circuits</td>
</tr>
</tbody>
</table>

- **Computer Architecture**
- **High-Level Synthesis**

- **DSP-oriented applications**
- **General-purpose code** (new applications and users)
Bridging the Gap Between Software and Hardware
Bridging the Gap Between Software and Hardware

HLS is still not meant for software programmers
Bridging the Gap Between Software and Hardware

HLS is still not meant for software programmers

```c
void(int* mem) {
    mem[512] = 0;
    for(int i=0; i<512; i++)
        mem[512] += mem[i];
}
```

(a) Unoptimized HLS Program; Execution Time = 27.236 clock cycles
Bridging the Gap Between Software and Hardware

HLS is still not meant for software programmers

```c
void(int* mem){
    mem[512] = 0;
    for(int i=0; i<512; i++)
        mem[512] += mem[i];
}
```

(a) Unoptimized HLS Program; Execution Time = 27.236 clock cycles

```c
// Width of MPort = 16 * sizeof(int)
#define ChunkSize (sizeof(MPort)/sizeof(int))
#define LoopCount (512/ChunkSize)
// Maximize data width from memory
void(MPort* mem){
    // Use a local buffer and burst access
    MPort buff[LoopCount];
    memcpy(buff, mem, LoopCount);
    // Use a local variable for accumulation
    int sum=0;
    for(int i=1; i<LoopCount; i++)
        // Use additional directives where useful
        // e.g. pipeline and unroll for parallel exec.
        #pragma PIPELINE
        for(int j=0; j<ChunkSize; j++)
            #pragma UNROLL
            sum+=((int)(buff[i]>>(j*sizeof(int)*8)));
    mem[512]=sum;
}
```

(b) Optimized HLS Program; Execution Time = 302 clock cycles
Bridging the Gap Between Software and Hardware

HLS is still not meant for software programmers

Develop new programming models for HW design
HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code
Bridging the Gap Between Software and Hardware

Sequential C-based synthesis still limits achievable parallelism

HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code
Bridging the Gap Between Software and Hardware

Sequential C-based synthesis still limits achievable parallelism

HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code

Elakhras, Guerrieri, Josipović, and Ienne. Unleashing Parallelism in Elastic Circuits with Faster Token Delivery. FPL 2022 Best Paper Award Nominee

Cheng, Josipović, Wickerson, and Constantinides. Dynamic Inter-Block Scheduling for HLS. FPL 2022 Best Paper Award Nominee
Bridging the Gap Between Software and Hardware

Sequential C-based synthesis still limits achievable parallelism

HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code

Sequential C-based synthesis still limits achievable parallelism

Develop compiler techniques to support irregular parallelism

Elakhras, Guerrieri, Josipović, and Ienne. Unleashing Parallelism in Elastic Circuits with Faster Token Delivery. FPL 2022 Best Paper Award Nominee

Cheng, Josipović, Wickerson, and Constantinides. Dynamic Inter-Block Scheduling for HLS. FPL 2022 Best Paper Award Nominee
Bridging the Gap Between Software and Hardware

- HLS is still not meant for software programmers
- HLS often fails in extracting parallelism from software code
- HLS circuits need hardware-level functional verification
Bridging the Gap Between Software and Hardware

**Functional verification** of HLS circuits using hardware simulation → inefficient and limited

- HLS is still not meant for software programmers
- HLS often fails in extracting parallelism from software code
- HLS circuits need hardware-level functional verification
Bridging the Gap Between Software and Hardware

HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

Functional verification of HLS circuits using hardware simulation $\rightarrow$ inefficient and limited
Bridging the Gap Between Software and Hardware

- HLS is still not meant for software programmers
- HLS often fails in extracting parallelism from software code
- HLS circuits need hardware-level functional verification

**Functional verification** of HLS circuits using hardware simulation → inefficient and limited

**Formal verification** for HLS: prove HLS correctness and improve HLS-generated circuits
Bridging the Gap Between Software and Hardware

**HLS is still not meant for software programmers**

**HLS often fails in extracting parallelism from software code**

**HLS circuits need hardware-level functional verification**

**Functional verification** of HLS circuits using hardware simulation → inefficient and limited

**Formal verification** for HLS: prove HLS correctness and improve HLS-generated circuits

---

Xu, Murphy, Cortadella, and Josipović. Eliminating excessive dynamism of dataflow circuits using model checking. Submitted to FPGA 2023.
Bridging the Gap Between Software and Hardware

**HLS is still not meant for software programmers**

**HLS often fails in extracting parallelism from software code**

**HLS circuits need hardware-level functional verification**

**Functional verification** of HLS circuits using hardware simulation → inefficient and limited

**Formal verification** for HLS: prove HLS correctness and improve HLS-generated circuits

Xu, Murphy, Cortadella, and Josipović. Eliminating excessive dynamism of dataflow circuits using model checking. Submitted to FPGA 2023.

Generic dataflow logic
Bridging the Gap Between Software and Hardware

HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

Functional verification of HLS circuits using hardware simulation \(\rightarrow\) inefficient and limited

Formal verification for HLS: prove HLS correctness and improve HLS-generated circuits

Customized dataflow logic

Xu, Murphy, Cortadella, and Josipović. Eliminating excessive dynamism of dataflow circuits using model checking. Submitted to FPGA 2023.
Bridging the Gap Between Software and Hardware

HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

Typical reconfigurable platforms are not suitable for HLS circuits
Bridging the Gap Between Software and Hardware

HLS is still not meant for software programmers

HLS often fails in extracting parallelism from software code

HLS circuits need hardware-level functional verification

Typical reconfigurable platforms are not suitable for HLS circuits

Programming generality and bit-level mapping of FPGAs → performance and power inefficiencies
Bridging the Gap Between Software and Hardware

Programming generality and bit-level mapping of FPGAs → performance and power inefficiencies

**FPGA-oriented** HLS optimizations and transformations

- HLS is still not meant for software programmers
- HLS often fails in extracting parallelism from software code
- HLS circuits need hardware-level functional verification
- Typical reconfigurable platforms are not suitable for HLS circuits
Bridging the Gap Between Software and Hardware

Programming generality and bit-level mapping of FPGAs → performance and power inefficiencies

**FPGA-oriented** HLS optimizations and transformations

- HLS is still not meant for software programmers
- HLS often fails in extracting parallelism from software code
- HLS circuits need hardware-level functional verification
- Typical reconfigurable platforms are not suitable for HLS circuits

Total delay pre-synthesis = 3 ns
Bridging the Gap Between Software and Hardware

- HLS is still not meant for software programmers
- HLS often fails in extracting parallelism from software code
- HLS circuits need hardware-level functional verification
- Typical reconfigurable platforms are not suitable for HLS circuits

Programming generality and bit-level mapping of FPGAs → performance and power inefficiencies

FPGA-oriented HLS optimizations and transformations

LUT delay = 0.5 ns

Total delay post-place-and-route = 1 ns
DYNAMO: Digital Systems and Design Automation Group

Enable diverse users to accelerate compute-intensive applications on hardware platforms

- High-level abstractions, programming languages, software applications
- HLS compilers, formal methods, electronic design automation
- Hardware accelerators, systems, digital design, computer architecture
Thanks! 😊

Research group: https://dynamo.ethz.ch/

Dynamatic HLS tool: https://dynamatic.epfl.ch/