High-Level Synthesis of Dynamically Scheduled Circuits

Lana Josipović

December 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 of program functionality and operation schedule with a train route from Lausanne, Gare to Le Säppi, Gare and back](image)
Standard HLS

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

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

![Diagram of program functionality and operation schedule]
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 the program functionality and operation schedule](Image)

Naïve schedule:

<table>
<thead>
<tr>
<th>Clock cycles</th>
<th>LD regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Operation schedule**

**Program functionality**

**Static controller**

2 stages
**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];
}
```

**Naïve schedule:**

```
LD x[i]  LD c[n-i]  mul
```

**Clock cycles**

1. Loop iterations
2. 1
3. 2
4. 3

**Program functionality**

**Operation schedule**

**Static controller**

2 stages
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 program functionality and operation schedule](image)

Naïve schedule:
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]
**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:**

```
LD x[i]    LD c[n-i]     +
*             +
< 2 stages

LD regs  mul  add
 mul  add
 mul  add
```

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

```c
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

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

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

Fork

Join

Branch

Merge

Branch
Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge

Dataflow Components

- Fork
- Join
- Branch
- Merge
Dataflow Components

Fork

Join

Branch

Merge
Dataflow Components

- Fork
- Join
- Branch
- Merge
Dataflow Components

- Fork
- Join
- Branch
- Merge
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];
}

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];
}
From Program to Dataflow Circuit

```c
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

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

Load ➔ Fork ➔ FIFO ➔ Store

Resource sharing

Mul 1
Mul 2
Mul 1/2

Reaping the benefits of dynamic scheduling

Out-of-order memory

Speculative execution

Load ➔ LSQ ➔ Memory

Save ➔ Speculator ➔ Commit

BB start ➔ Fork ➔ + ➔ Commit
Inserting Buffers

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

Buffers as registers to break combinational paths

Josipović, Sheikhha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
Rizzi, Guerrieri, Ienne, and Josipović. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. FPL 2022
Inserting Buffers

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

Buffers as FIFOs to regulate throughput
Inserting Buffers

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

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

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

NOW (with buffers)

Start: i=0

Merge

Fork

LD x[i]

FIFO

LD hist[x[i]]

LD weight[i]

Fork

N <

4 stages

Branch

ST hist[x[i]]

Exit: i=N

BEFORE (without buffers)

Start: i=0

Merge

Fork

LD x[i]

LD hist[x[i]]

LD weight[i]

Fork

1 comb.

Exit: i=N

4 stages

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

Rizzi, Guerrieri, Ienne, and Josipović. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. FPL 2022
Model each program loop as a concurrent, choice-free Petri net (= marked graph)

- MILP model to optimize throughput of the choice-free Petri net under a clock period constraint
Optimizing Performance

- Model each program loop as a **concurrent, choice-free Petri net (= marked graph)**
  - **MILP model** to optimize throughput of the choice-free Petri net under a clock period constraint

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

\[ \max: \Phi - \lambda \cdot \sum_{c} N_c \]

throughput small const. buffer slots

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

- Model each program loop as a **concurrent, choice-free Petri net (= marked graph)**
  - **MILP model** to optimize throughput of the choice-free Petri net under a clock period constraint

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

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

![Diagram of a Petri net model with buffer placement and timing constraints]

Target CP = 4 ns

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

\[
\text{target period } \ 	ext{N-buff}
\]

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

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

in/out arrival time \hspace{1cm} \text{unit comb. delay}
Optimizing Performance

- Model each program loop as a **concurrent, choice-free Petri net (= marked graph)**
  - MILP model to optimize throughput of the choice-free Petri net under a clock period constraint

---

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

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

- $\text{max: } \Phi - \lambda \sum_c N_c$
- $\text{target period } \text{N-buff } t_{c,\text{out}} \geq t_{c,\text{in}} - CP \cdot R_c$
- $CP \geq t_{c,\text{in}} \geq t_{c,\text{out}} + D_u$

\(t_{c,i}\) in/out arrival time  \(D_u\) 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

- Model each program loop as a **concurrent, choice-free Petri net (= marked graph)**
  - **MILP model** to optimize throughput of the choice-free Petri net under a clock period constraint

**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

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

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

\[
CP \geq t_{c_2}^c \geq t_{c_1}^c + D_u
\]

**token retiming**

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

**channel occupancy**

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

- Model each program loop as a **concurrent, choice-free Petri net (= marked graph)**
- **MILP model** to optimize throughput of the choice-free Petri net under a clock period constraint

**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

Throughput:

\[ \Phi = 1 \]

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

• Model each program loop as a concurrent, choice-free Petri net (= marked graph)
  — MILP model to optimize throughput of the choice-free Petri net under a clock period constraint

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 \\

\forall c \in C:
\begin{align*}
& t^\text{out}_c \geq t^\text{in}_c - CP \cdot R_c \\
& CP \geq t^\text{in}_{c_1} \geq t^\text{out}_{c_1} + D_u \\
\end{align*}
\]

Token retiming:
\[
\theta_c = B_b + r_v - r_u \\
\theta_c \geq \Phi + R_c - 1
\]

channel occupancy

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

• Model each program loop as a concurrent, choice-free Petri net (= marked graph)
  — MILP model to optimize throughput of the choice-free Petri net under a clock period constraint

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

\[
\max: \Phi - \lambda \cdot \sum_c N_c \\
\text{target CP} = 4 \text{ ns} \\
\text{throughput: } \Phi = 1
\]
Optimizing Performance

- Model each program loop as a **concurrent, choice-free Petri net (= marked graph)**
  - **MILP model** to optimize throughput of the choice-free Petri net under a clock period constraint

**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

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

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

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

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

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

• Model each program loop as a concurrent, choice-free Petri net (= marked graph)

- MILP model to optimize throughput of the choice-free Petri net under a clock period constraint

**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

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

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

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

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

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

---

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

- Model each program loop as a **concurrent, choice-free Petri net (= marked graph)** — **MILP model** to optimize throughput of the choice-free Petri net under a clock period constraint

**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

\[
\begin{align*}
& \max: \Phi - \lambda \cdot \sum_c N_c \\
& t^\text{out}_c \geq t^\text{in}_c - CP \cdot R_c \\
& CP \geq t^\text{out}_c^{c_2} \geq t^\text{out}_c^{c_1} + D_u \\
& \theta_c = B_b + r_v - r_u \\
& \theta_c \geq \Phi + R_c - 1 \\
& N_c \geq \theta_c + \theta^o_c
\end{align*}
\]
Optimizing Performance

- Model each program loop as a **concurrent, choice-free Petri net (= marked graph)**
  - **MILP model** to optimize throughput of the choice-free Petri net under a clock period constraint

**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

Target CP = 3 ns
Optimizing Performance

• Model each program loop as a **concurrent, choice-free Petri net (= marked graph)**
  — **MILP model** to optimize throughput of the choice-free Petri net under a clock period constraint

**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

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

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

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

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

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

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

- Model each program loop as a **concurrent, choice-free Petri net (= marked graph)**

  - **MILP model** to optimize throughput of the choice-free Petri net under a clock period constraint

**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

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

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

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

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

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

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

Target CP = 3 ns

Throughput: \( \Phi = 1/2 \)

Josipović, Sheikha, Guerrieri, Ienne, and Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. FPGA 2020 Best paper award
HLS of Dynamically Scheduled Circuits

Catching up with static HLS

Pipelining

Resource sharing

Out-of-order memory

Speculative execution
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;
}
```

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

```plaintext
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 choice-free Petri net model 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;
}
```

Units underutilized (low throughput)

Sharing possible without damaging throughput

Use choice-free Petri net model to decide what to share

Josipović, Marmet, Guerrieri, and Ienne. Resource Sharing in Dataflow Circuits. FCCM 2022. **Best Paper Award Nominee**
Inserting Buffers

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

Backpressure from slow paths prevents pipelining
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

RAW dependency not honored!

What about memory?
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
We Need a Load-Store Queue (LSQ)!

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

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

How to supply program order to the LSQ?

```plaintext
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

load x[i]  
load y[i]  
store x[i]
```

```plaintext
load x[0]
```

Ordering (load-store queue)

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

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

- Dataflow circuits have no notion of 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
```

**Dynamic knowledge of basic block sequence from the dataflow circuit**
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
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

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

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

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

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

But... dataflow computation is resource-expensive!
The Cost of Dataflow Computation

for (i=0; i<N; i++) {
    a[i] = a[i]*c;
}
The Cost of Dataflow Computation

Distributed dataflow handshake mechanism: resource and frequency overhead
The Cost of Dataflow Computation

Do we need expensive dataflow logic *everywhere*?
Removing Excessive Dynamism

Data is never stalled

Possible stall
Removing Excessive Dynamism

Data is never stalled

Restrict the generality of dataflow logic whenever it is not needed
Removing Excessive Dynamism

How to guarantee correctness of simplifications for any possible circuit behavior?

Data is never stalled

Possible stall
Proving Properties to Eliminate Excessive Dynamism

For each channel: prove the **absence of backpressure** (remove logic to compute the ready signal)

\[ \text{AG (valid} \rightarrow \text{ready)} \]
For each channel: prove the **absence of backpressure**
(remove logic to compute the ready signal)
\[ \text{AG (valid} \rightarrow \text{ready)} \]

For each pair of channels: prove **trigger equivalence**
(remove logic to compute one of the valid signals)
\[ \text{AG (valid1} \leftrightarrow \text{valid2)} \]
Proving Properties to Eliminate Excessive Dynamism

For each channel: prove the absence of backpressure (remove logic to compute the ready signal)
AG (valid → ready)

For each pair of channels: prove trigger equivalence (remove logic to compute one of the valid signals)
AG (valid1 ↔ valid2)

Up to 50% area reduction without a performance penalty

But it is very slow (~hrs)...

Xu, Murphy, Cortadella, and Josipović. Eliminating excessive dynamism of dataflow circuits using model checking. FPGA 2023.
Ensuring Scalability by Compositional Verification

- **Decompose circuit** into regions whose properties can be verified independently
- **Abstract the complexity** of other regions into simpler nodes that have the same properties as the circuit they encapsulate

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

Xu, Murphy, Cortadella, and Josipović. Eliminating excessive dynamism of dataflow circuits using model checking. FPGA 2023.
Ensuring Scalability by Compositional Verification

- **Decompose circuit** into regions whose properties can be verified independently
- **Abstract the complexity** of other regions into simpler nodes that have the same properties as the circuit they encapsulate

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

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

Up to 8X reduction in checking time

Xu, Murphy, Cortadella, and Josipović. Eliminating excessive dynamism of dataflow circuits using model checking. FPGA 2023.
Enable diverse users to accelerate compute-intensive applications on hardware platforms
MSc & BSc Projects and Theses

• Use **Petri nets** to describe circuits and their behaviors
  – Component modelling
  – Performance and area optimizations

• Use **model checking** to prove circuit properties and improve their quality
  – Checking more complex properties
  – Dealing with scalability issues

• And many other topics...

• Check link on last slide for (non-exhaustive) list of projects!

Come work with us! 😊
New Course in Spring 2023: Synthesis of Digital Circuits

- Algorithms, tools, and methods to generate circuits from high-level programs
  - How does ‘classic’ HLS work?
- Recent advancements and current challenges of HLS for FPGAs
  - What is HLS still missing?
- Course organization
  - First part: lectures+exercises
  - Second part: practical work + seminar-like discussions
- Link to Course Catalogue info (2023)

Hope to see you there! 😊
Thanks! 😊

Research group

Project list 2023

Link

Link