From C/C++ to Dynamically Scheduled Circuits

Lana Josipović

September 2022
How to perform hardware design?
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];
}
```

![Train schedule diagram](image)

- **Program functionality**
- **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];
}
```

![Diagram showing 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

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

Naïve schedule:

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

Clock cycles

```
C1  C2  C3  C4  C5  C6  C7  C8  C9  C10  C11  C12
1   2   3   4   5   6   7   8   9   10   11   12
```

Program functionality

Operation schedule

Static controller

2 stages

N
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

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

---

**Naïve schedule:**

<table>
<thead>
<tr>
<th>Loop iterations</th>
<th>Clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12</td>
</tr>
</tbody>
</table>

**Static controller**

- LD x[i]
- LD c[n-i]
- *
- <
- +
- Static controller

Program functionality

Operation schedule

---

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]
* 2 stages
+<
```

Program functionality

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];
}
```

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];
}
```

![Diagram of program functionality and operation schedule]

- **Naive schedule:**
- **Pipelined schedule:**

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

• 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];
}

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
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

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**
  - Load
  - Fork
  - FIFO
  - Store
  - Pipelines

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

Reaping the benefits of dynamic scheduling

- **Out-of-order memory**
  - Load
  - Store
  - LSQ
  - Memory

- **Speculative execution**
  - Save
  - Speculator
  - Fork
  - Commit
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

- Fork
- Join
- Branch
- Merge
Dataflow Components

- Fork
- Join
- Branch
- Merge

Symbols:
- (+) for addition
- (*) for multiplication
- STORE for storage
Dataflow Components

- Fork
- Join
- Merge
- Branch
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];
}
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

```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

Start: $i=0$

Merge

Fork

LD $x[i]$

LD weight[$i$]

ST $hist[x[i]]$

Fork

LD $hist[x[i]]$

Branch

$+$

4 stages

$<$

Exit: $i=N$

1 comb.

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

Out-of-order memory

Speculative execution
Inserting Buffers

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

Buffers as registers to break combinational paths
Inserting Buffers

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

Buffers as FIFOs to regulate throughput

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

BEFORE (without buffers)

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

Fork

LD $\text{hist}[x[i]]$

Fork

LD $\text{weight}[i]$

ST $\text{hist}[x[i]]$

1 comb.

Exit: $i=N$

4 stages

Branch

BEFORE (without buffers)

Start: $i=0$

Merge

Fork

LD $x[i]$

Fork

LD $\text{hist}[x[i]]$

Fork

LD $\text{weight}[i]$

ST $\text{hist}[x[i]]$

1 comb.

Exit: $i=N$

4 stages

Branch

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]

M

Fork

LD weight[i]

R

Merge

Fork

LD hist[x[i]]

Fork

ST hist[x[i]]

FIFO

FIFO

FIFO

Branch

Exit: i=N

Mixed integer linear programming (MILP) model based on Petri net theory
- Analyze token flow through the circuit
- Determine buffer placement and sizing
- Maximize throughput for a target clock period

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
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
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

```java
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

```cpp
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 MILP (performance optimization) information to decide what to share

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

```plaintext
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

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
We Need a Load-Store Queue (LSQ)!

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

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

```assembly
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

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

Josipović, Bhattacharrya, Guerrieri, and Ienne. Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs. FPT 2019
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];
}
```

Josipović, Bhattacharrya, Guerrieri, and Ienne. Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs. FPT 2019
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
  - 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
  - Load
  - FIFO
  - Store
  - LSQ

- Speculative execution
  - Save
  - Speculator
  - Fork
  - Commit
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;

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);
Long control flow decision prevents pipelining
Nonspeculative vs. Speculative System

Nonspeculative schedule

Speculative schedule
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
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
Speculative Dataflow Circuit

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

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

Start, i=0

Merge

Buff

Fork

Load a[i]

Load b[i]

Save

Spec. Branch

End

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]

Save

Spec. Branch

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

Output boundary: Commit units
Speculative Dataflow Circuit

Start, i=0

Merge

Buff

Fork

Load[1][i]

Load[2][i]

Fork

1

i

Merge

Buff

Fork

Load[1][i]

Load[2][i]

Fork

1

i

Before (without speculation)

Wait for long-latency condition

Continue computing before condition known

BEFORE (without speculation)
Speculative Dataflow Circuit

High-throughput speculative pipeline
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

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.
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.
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

![Diagram](image)

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.

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)**
Thanks! ☺

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

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