Dataflow Design

Montek Singh
UNC Chapel Hill

ASYNC 2024 Summer School
Week 1: July 1
Outline

* Dataflow basics
  * Pipelining primitives

* Performance estimation
  * “Canopy Graph” analysis
Dataflow basics
What is Dataflow?

• Graphical description of operations in a computation

• Sequencing is determined by data dependencies
  • inputs trigger a function
  • ... instead of an overall control structure

• Intuitive, natural representation for:
  • data-driven algorithms, e.g. DSPs
  • stream processing

• Implementation is not necessarily asynchronous
  • but async is often a natural match
Example: multiply-accumulate

Motivation: linear algebra core operation

\[ y \leftarrow \alpha x + y \] (SAXPY)

If you care about DSP, HPC, AI/deep learning... this is a useful kernel to implement

[Original slide by Benjamin Hill]
Dataflow primitives

* Reading from all input channels, writing to all output channels

* Reading from 1, writing to one-of-\(N\) (demux)

* Reading from one-of-\(N\), writing to 1 (mux / conditional merge)

* Other misc useful blocks:
  - initialization
  - source/sink
  - merging/arbitration

[Modified from original slide by Benjamin Hill]
BUFFER

* Transmit token from input to output with storage and handshaking flow control
  - one pipeline stage (FIFO stage)
  - latch + handshake control

Also known as: slack buffer, one-place FIFO, latch

*[In?x; Out!x]*
FORK / COPY

Copy input token to multiple destinations

Also known as: n-way link

*[In?x; Out₀!x, …, Outₙ₋₁!x]*

[Original slide by Benjamin Hill]
JOIN / FUNCTION

Read values from all inputs, compute result and send on output

Example functions: arithmetic, logic, decoding, etc.

Also known as: OPERATOR

[Original slide by Benjamin Hill]
**Multiplexer (MUX)**

Select one input to send to output based on control signal

- ignore other input (do not consume)
- generalizable to \( N \) inputs

Not to be confused with combinational MUX:

- same basic behavior, but this is a dataflow operator
- unused input channel is not consumed; its data is still available

Also known as: controlled merge, conditional join

[Original slide by Benjamin Hill]
DEMUX

Steer/route input to one of two outputs
- based on value of control signal
- generalizable to $N$ outputs

Also known as: SPLIT

*[In?x, C?c;
  [ c=0 -> Out₀!x
  [] c=1 -> Out₁!x
  ]
]
Initial token buffer

Send one initial value token, then behave as a normal buffer

Also known as: INITIALIZER

Out!value; *[In?x; Out!x]
Repeatedly send tokens with same constant value

Also known as: bit/token generator
SINK

Consume and discard input token

- Not particularly useful by itself, but in combination with other dataflow primitives

Also known as: (bit) bucket

*[In?value]*

[Original slide by Benjamin Hill]
Uncontrolled merge

Combine two input streams to one output

Depending on system design, selection is either:

- **deterministic** – only one input will arrive at a time (ensured by environment)
- **non-deterministic** – requires arbitration to choose if both inputs can arrive close together

Also known as: MIXER, JOIN
Recap: Dataflow primitives

* Reading from all input channels, writing to all output channels
* Reading from 1, writing to one-of-\(N\) (demux)
* Reading from one-of-\(N\), writing to 1 (mux / conditional merge)

* Other misc useful blocks:
  - initialization
  - source/sink
  - merging/arbitration

[Modified from original slide by Benjamin Hill]
Some useful design patterns
Wagging or Multithreading

Problem: Slow function block

Solution: Duplicate function block and interleave data between them

→ Improves throughput at the cost of area

Example: large arithmetic block where it is difficult to add internal pipelining

Not just for compute, could also be storage (e.g. tree FIFO)
Resource sharing

Idea: share one expensive or unique resource between multiple users

Improves area at the cost of throughput
IF statement

Useful for high-level synthesis

Shown with FUNCTION blocks but can also be other dataflow graphs (e.g. nested IF statements)
WHILE loop

Can also implement other loop constructs with a similar pattern
Performance Estimation
Performance Basics: pipeline stages

Each stage characterized by three delays:

- **Forward latency**, $L_f$
  - Time for data to propagate forward

- **Reverse latency**, $L_r$
  - Time for a stage to receive and process ack
  - Time for a ‘hole’ to travel backward

- **Cycle time**, $T = L_f + L_r$
  - Throughput, $tpt = 1 / \text{cycle time}$
Goal

Motivation: crucial part of an optimizing design flow

- Used repeatedly in an optimization loop
- Requires low runtime and good accuracy
Early work: Pipeline Rings

Classic work by T. Williams and M. Horowitz [ISSCC-91]

Ring throughput depends on its occupancy (#items)

- For small number of items: under-utilization limits throughput
- For small number of holes: congestion limits throughput
- Throughput also limited by the slowest stage
- Graph is a convex shape: “Canopy Graph”
  - [term coined by Singh et al. ASYNC-02 and Gill/Singh ICCAD-08]
**Canopy Graphs for linear pipeline**

*Canopy graph: also useful approximation for linear pipelines*

- **In steady state:** linear pipeline can be modeled as ring
  - Rate at which data enters and leaves is identical
  - *i.e.* one token leaves ➔ one token enters
Key Idea: Generalize Canopy Graphs

* Goal: Find the **system-level throughput** for an async dataflow system
  - Use a modular, “divide-and-conquer” method

* Challenge: Throughput is not composable
  - Complex interdependencies dictate throughput

* Take problem to higher dimension to make decomposable
  - One-dimensional throughput is not composable
  - Two-dimensional throughput-occupancy pairs are

![Diagram showing throughput and occupancy](image)
Performance Analysis: Method

- **Modular method for performance analysis**
  - Exploits system hierarchy with “divide-and-conquer” method
  - First: calculate canopy graph at each leaf node
    - Each leaf node is a single stage
  - Next: compose canopy graphs at each level of the hierarchy
  - Finally: canopy graph for root node gives system-level performance

- **Requires composition algorithm for common circuit structures**
  - Parallel, sequential, conditional, and iterative

---

1) Parallel Composition

Parallel structures [Lines98]
- Data copied at fork
- A and B compute in parallel
- Results recombined at join

Operation invariants under composition:
1) # of items in each branch equal
2) Branches have same throughput

Throughput of structure:
- Intuition: at each occupancy, throughput limited by slower branch
  ⇒ Intersection of canopy graphs of A and B
2) Sequential Composition

Sequential Structures [Lines98]
- Data transmitted through A, then through B

Operation invariants under composition:
1) Find total # items: sum of # items in both pipes
2) Throughput $A = \text{Throughput } B$
3) Max throughput: limited by slower pipeline

Throughput of structure:
⇒ "horizontal sum" of canopy graphs of A and B
- At each throughput, add the occupancies of the two pipelines
3) Conditional Composition

Operation invariants under composition:

- Ratio of # items in each branch = ratio of probabilities
  \[
  \frac{\text{Occupancy}_0}{p_0} = \frac{\text{Occupancy}_1}{p_1}
  \]

- Ratio of throughput of each branch = ratio of probabilities
  \[
  \frac{TPT_0}{p_0} = \frac{TPT_1}{p_1}
  \]

Throughput of conditional structure:

- Divide each branch’s canopy graph by its probability \( p_i \)
- Compute intersection of scaled canopy graphs

“Bursty” inputs cause additional bottlenecks (see ICCAD-08 paper for details)

Example:

- \( p_0 = 2/3 \) and \( p_1 = 1/3 \)
- 2 items enter \( \text{branch}_0 \)
- 1 item enters \( \text{branch}_1 \)
3) Conditional Composition (cont’d)

**Step 1) uniform scaling:** enlarge each branch’s canopy graph

Example: $p_0 = \frac{2}{3}$ and $p_1 = 1 - p_0 = \frac{1}{3}$

**Step 2) intersection:** finds system-level performance

![Graph showing uniform scaling and intersection for branches 0 and 1](image-url)

- **Branch0 Performance**
  - Assume $p_0 = \frac{2}{3}$
  - Original canopy graph
  - Scaled by dividing by $\frac{2}{3}$

- **Conditional composition**
  - Scaled by dividing by $\frac{1}{3}$

- **Branch1 Performance**
  - Assume $p_1 = \frac{1}{3}$
  - Original canopy graph
  - Scaled by dividing by $\frac{1}{3}$
4) Iterative Loop Composition

- **Operation invariants under composition:**
  - Each item passes through the loop multiple times
  - Loop can handle multiple items simultaneously

- **Throughput of composition**
  - # data items processed decreases as iteration count increases

⇒ Scale down based on expected number of iterations

---

**Throughput vs. Occupancy with Expected Iterations = 3.33**

- Divide by 3.33
- Loop composition
- Loop body
- Occupancy
- Throughput
Analysis: Benchmark Examples

- **Analysis algorithm demonstrated on 8 benchmarks**
  - Chosen to represent a variety of circuit constructs

<table>
<thead>
<tr>
<th>Example</th>
<th>Composition Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Parallel</td>
</tr>
<tr>
<td>CORDIC</td>
<td>✔</td>
</tr>
<tr>
<td>CRC</td>
<td></td>
</tr>
<tr>
<td>DIFFEQ</td>
<td>✔</td>
</tr>
<tr>
<td>GCD</td>
<td></td>
</tr>
<tr>
<td>Ray-tracing</td>
<td></td>
</tr>
<tr>
<td>MULT</td>
<td>✔</td>
</tr>
<tr>
<td>JPEG</td>
<td>✔</td>
</tr>
</tbody>
</table>

- **Evaluated several circuit implementations of some**
  - Naive implementation vs. hand-optimized version
  - Different choice models: uniform random vs. correlated
**Performance Analysis: Results**

* Total of 12 different circuit examples tested
  * Error < 4% for all examples, runtime ≤ 10 ms for all examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Version</th>
<th>Size</th>
<th>Throughput</th>
<th>Error (%)</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># stages</td>
<td>simulated</td>
<td>predicted</td>
<td></td>
</tr>
<tr>
<td><strong>CORDIC</strong></td>
<td>original</td>
<td>31</td>
<td>90.9</td>
<td>90.9</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>optimized</td>
<td>44</td>
<td>167</td>
<td>167</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>bursty inputs</td>
<td>44</td>
<td>83</td>
<td>83</td>
<td>0.00</td>
</tr>
<tr>
<td><strong>CRC</strong></td>
<td>original</td>
<td>23</td>
<td>292</td>
<td>286</td>
<td>2.05</td>
</tr>
<tr>
<td></td>
<td>optimized</td>
<td>27</td>
<td>352</td>
<td>357</td>
<td>1.42</td>
</tr>
<tr>
<td></td>
<td>bursty inputs</td>
<td>27</td>
<td>305</td>
<td>300</td>
<td>1.64</td>
</tr>
<tr>
<td><strong>DIFFEQ</strong></td>
<td>original</td>
<td>10</td>
<td>18.3</td>
<td>18.2</td>
<td>0.55</td>
</tr>
<tr>
<td><strong>GCD</strong></td>
<td>original</td>
<td>21</td>
<td>49</td>
<td>50</td>
<td>2.04</td>
</tr>
<tr>
<td><strong>Ray-</strong></td>
<td>original</td>
<td>21</td>
<td>161</td>
<td>167</td>
<td>3.73</td>
</tr>
<tr>
<td><strong>tracing</strong></td>
<td>optimized</td>
<td>166</td>
<td>222</td>
<td>222</td>
<td>0.00</td>
</tr>
<tr>
<td><strong>MULT</strong></td>
<td>original</td>
<td>13</td>
<td>38.7</td>
<td>38.4</td>
<td>0.78</td>
</tr>
<tr>
<td></td>
<td>optimized</td>
<td>21</td>
<td>167</td>
<td>167</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Performance Analysis: Summary

- **Fast:** restriction to hierarchical systems yielded fast runtimes
  - Divide-and-conquer approach with linear runtime
  - Modular canopy graph analysis for many constructs
    - Sequential, parallel, conditional, and loop
  - Expressive subset: modeled real-world applications
    - CORDIC, CRC, ray intersection algorithm, etc.

- **Accurate:** tested on several many non-trivial examples
  - Throughput estimates within 4% of simulation results
References


