DFG Performance Modeling and Transformations

We indicated earlier that Data Flow graphs (DFGs) are **untimed**, i.e., our analysis did not model the amount of time needed to complete a computation

In this lecture, we describe how to use DFGs for performance analysis

Performance estimation will be accomplished by modeling only two components: *actors* and *queues*

Once our new modeling constructs are introduced, we then turn our attention to *transformations* designed to enhance performance

Input sample rate is the time interval between two adjacent input samples from a data stream

For example, a digital sound system generates 44,100 samples per second

Input sample rate defines a **design constraint** for the *real-time* performance of the Data Flow system

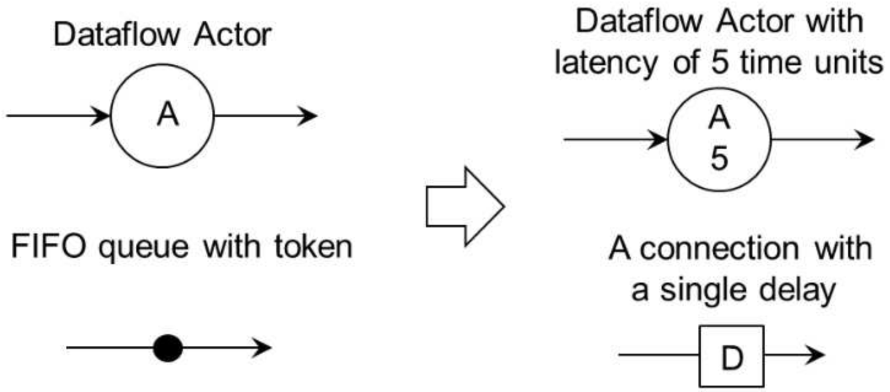
Similar constraints usually exists for *output sample rate*

Definitions

We use two common metrics as measures of performance:

- **Throughput**: the number of *samples* processed per second Note that input and output throughput may be different
- **Latency**: The time required to process a single token from input to output

The Data Flov



We used the symbols on the left earlier to model DFGs

For performance modeling, we

- Include a number within the *actor* symbol to model execution latency
- Replace FIFO *queues* with a communication channel, which includes delays

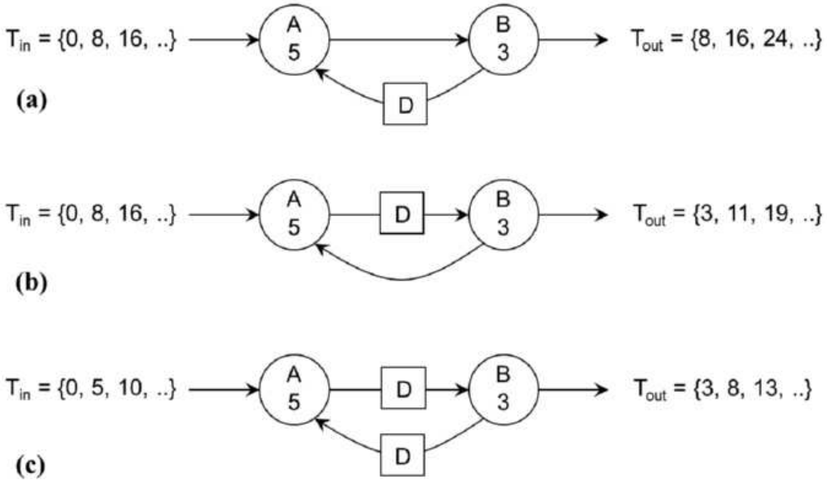
Definitions

Note that the number included in an *actor* represents the amount of time it takes (in clock cycles, nanoseconds, etc) **after** it fires

Time spent while waiting for input data is not counted

Also note that the *delay* element (which replaces FIFO *queues*) can hold exactly one token

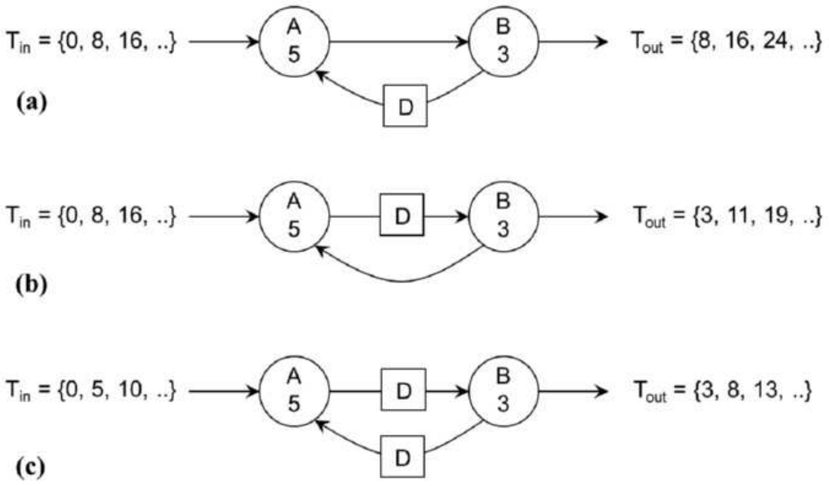
Think of *delay* elements as buffers with 1 unit of delay


T_{in} = {0, 8, 16, ..}
A 5
B 3
D
T_{out} = {8, 16, 24, ..}
(a)
T_{in} = {0, 8, 16, ..}
A 5
D
B 3
T_{out} = {3, 11, 19, ..}
(b)
T_{in} = {0, 5, 10, ..}
A 5
D
B 3
D
T_{out} = {3, 8, 13, ..}
(c)

We can use a performance annotated DFG to evaluate its execution time

In (a), (b) and (c) above, *actor* A introduces 5 units of latency while B introduces 3 units

Performance Analysis


T_{in} = {0, 8, 16, ..}
A 5
B 3
D
T_{out} = {8, 16, 24, ..}
(a)
T_{in} = {0, 8, 16, ..}
A 5
D
B 3
T_{out} = {3, 11, 19, ..}
(b)
T_{in} = {0, 5, 10, ..}
A 5
D
B 3
D
T_{out} = {3, 8, 13, ..}
(c)

The time stamp sequences on the left and right indicate when input samples are read and when output samples are produced

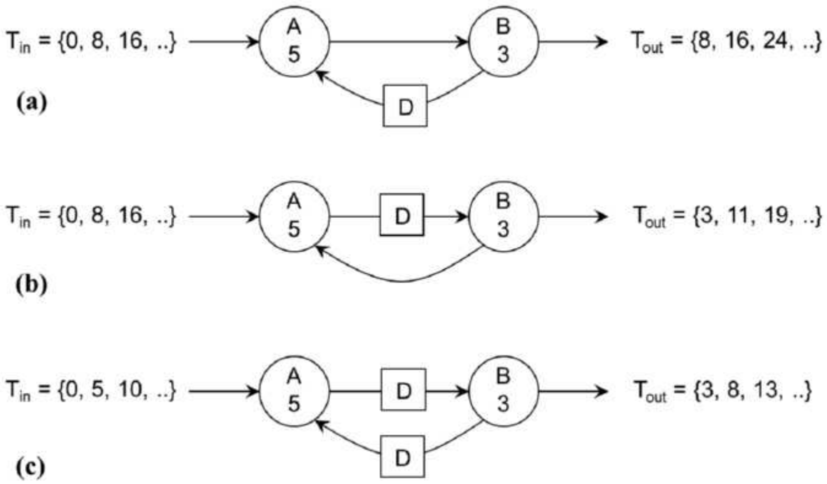
The time stamps for DFG (a) and (b) are different because of the position of the *delay* element in the loop

(a)requires the sum of execution times of A and B before producing a result

(b)can produce a result at time stamp 3 because the *delay* element allows it to execute immediately at system start time (we refer to this as *transient* behavior)

In this case, the *delay* elements affect only the latency of the first sample

Performance Analysis


T_{in} = {0, 8, 16, ..}
A 5
B 3
D
T_{out} = {8, 16, 24, ..}
(a)
T_{in} = {0, 8, 16, ..}
A 5
D
B 3
T_{out} = {3, 11, 19, ..}
(b)
T_{in} = {0, 5, 10, ..}
A 5
D
B 3
D
T_{out} = {3, 8, 13, ..}
(c)

In contrast, (c) shows that *delay* elements can be positioned to enable parallelism, and affect both latency and throughput

Both *actors* can execute in parallel in (c), resulting in better performance than (a) and (b)

The throughput of (a) and (b) is 1 sample per 8 time units, while (c) is 1 sample per 5 time units

Similar to a pipelined system, the throughput in (c) is ultimately **limited** to the speed of the **slowest** *actor* (A in this case -- B is forced to wait)

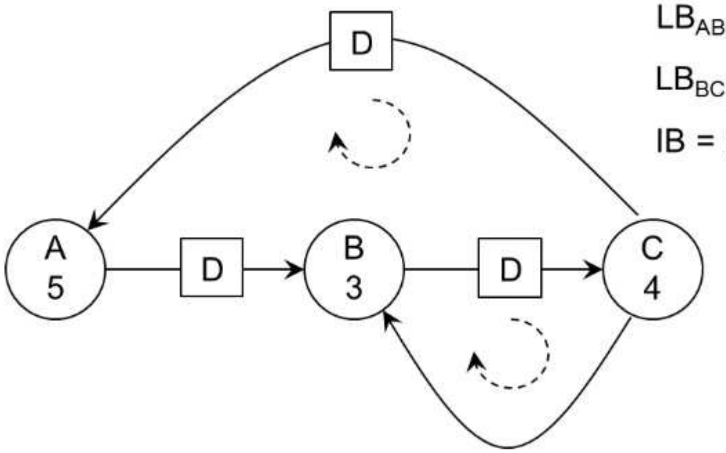
Limits on Throughput

As indicated, the distribution of the *delay* elements in the loops impacts performance

As an aid in analyzing performance, let's define

- ***Loop bound*** as the round-trip delay of a loop, divided by the number of delays in the loop
- ***Iteration bound*** as the *largest* loop bound in any loop of a DFG

Iteration bound defines an upper limit on the best throughput of a DFG



LB_{ABC} = (5+3+4)/3

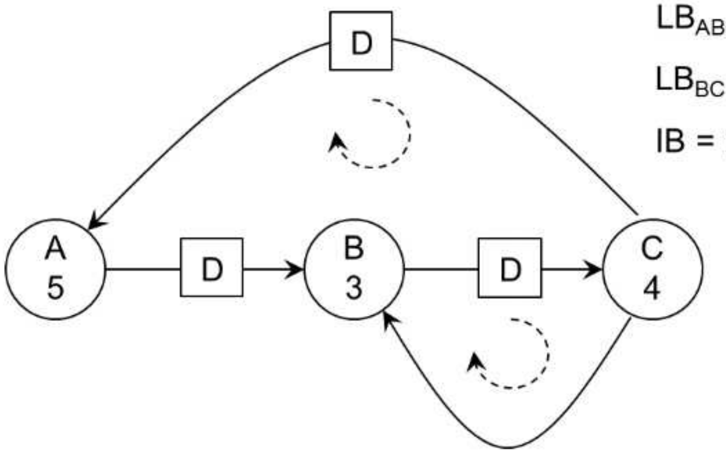
LB_{BC} = (3+4)/1

IB = max(LB_{ABC},LB_{BC})= 7

The *loop bounds* in this example are given as $LB_{BC} = 7$ and $LB_{ABC} = 4$

The *iteration bound* is 7 -- therefore, we need at least 7 time units per iteration

Limits on Throughput



$LB_{ABC} = (5+3+4)/3$

$LB_{BC} = (3+4)/1$

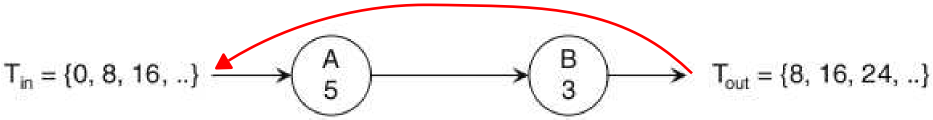
$IB = \max(LB_{ABC}, LB_{BC}) = 7$

From the graph, it is clear that loop BC is the bottleneck

- Note that *actors* A and C have *delay* elements on their inputs so they can operate in parallel

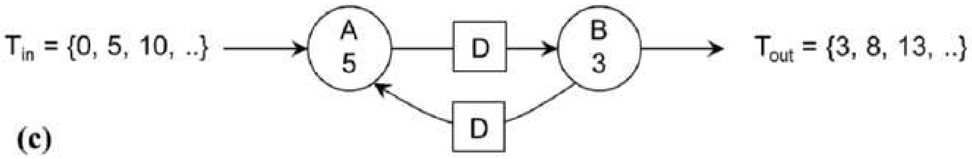
- On the other hand, *actor* B needs to wait for the result from C before it can *fire*
  - The missing *delay* element forces *actors* B and C to run sequentially

Note that *linear* graphs have implicit feedback loops that must be considered



Limits on Throughput

Also note that the iteration bound is an **upper limit** on throughput, and in reality, the DFG may **not** be able to achieve this throughput



The DFG above (from an earlier slide) has an iteration bound $(5 + 3)/2 = 4$ time units, but the throughput is limited to the *slowest actor* at 1 sample per 5 time units

A nice way to think about *actors* and *delays* is to consider an *actor* as a **combinational** circuit and a *delay* as a **buffer** or **pipeline** stage

Performance-Enhancing Transformations

Based on previous discussions, intuitively, it should be possible to ’tune’ the DFG to enhance performance, while maintaining the same functionality

Enhancing performance either *reduces latency* or *increases throughput* or both

The following transformations will be considered:

- **Multi-rate Expansion**: A transformation which converts a multi-rate synchronous DFG to a single-rate synchronous DFG
- **Retiming**: A transformation that redistributes the *delay* elements in the DFG Retiming changes the throughput but does **not** change the latency or the transient behavior of the DFG
- **Pipelining**: A transformation that introduces new *delay* elements in the DFG Pipelining changes both the throughput and transient behavior of the DFG
- **Unfolding**: A transformation designed to increase parallelism by duplicating *actors* Unfolding changes the throughput but **not** the transient behavior of the DFG

Multi-rate Transformation

The following is a systematic approach to transform a *multi-rate* DFG to a *single-rate* DFG:

- Determine the PASS *firing rates* of each *actor*

- Duplicate each *actor* the number of times indicated by its *firing rate*

  For example, if *actor A* has a *firing rate* of 2, create duplicate *actors* A0 and A1

- Convert each **multi-rate** *actor* input/output to multiple **single-rate** input/outputs For example, an *actor* with an input consumption rate of 3 is replaced with 3 single-rate inputs

- Re-wire the *queues* in the DFG to connect all *actors*

- Re-introduce the initial *tokens* in the DFG, distributing them sequentially over the single-rate *queues*

Multi-rate Transformation

The following DFG shows *actor* A produces **three** tokens per firing, and *actor* B consumes **two** tokens per firing

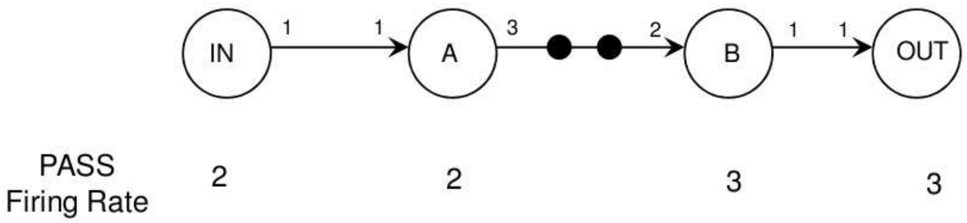
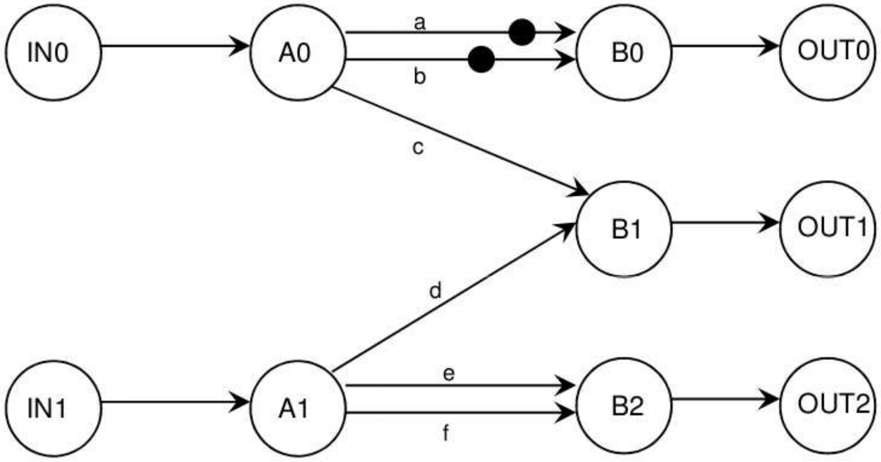


Fig. 2.28 Multi-rate data flow-graph

After completing the steps above, we obtain the following DFG



The initial tokens are redistributed in order *a*, *b*, etc

Fig. 2.29 Multi-rate SDF graph expanded to single-rate

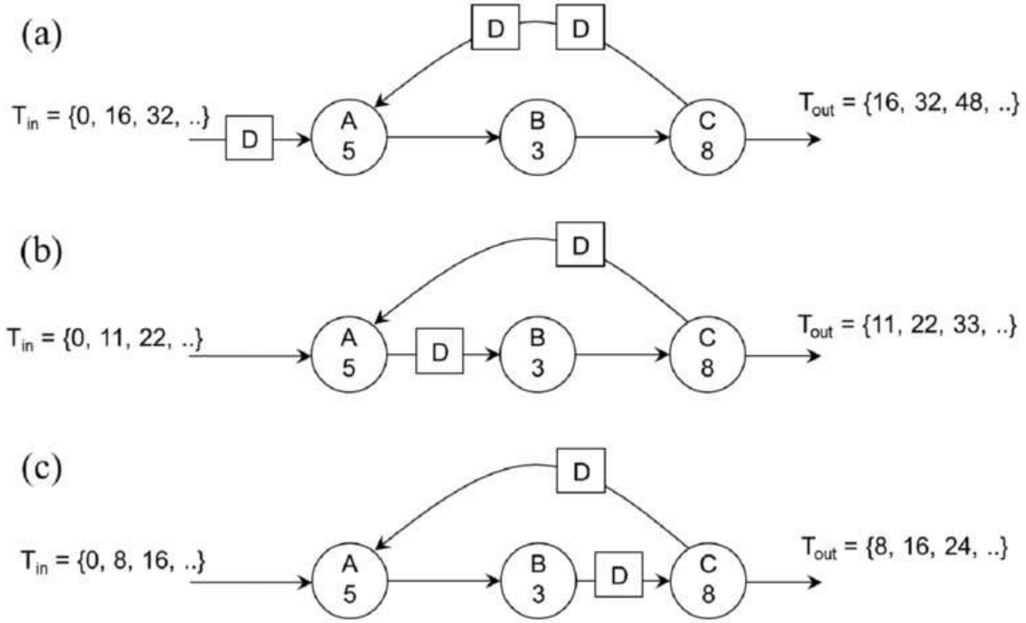
Here, the *actors* are **duplicated** according to their *firing rates*, and all *multi-rate* I/O are converted to *single-rate* I/O

Retiming Transformation

Retiming **redistributes** *delay* elements in the DFG as a mechanism to increase throughput

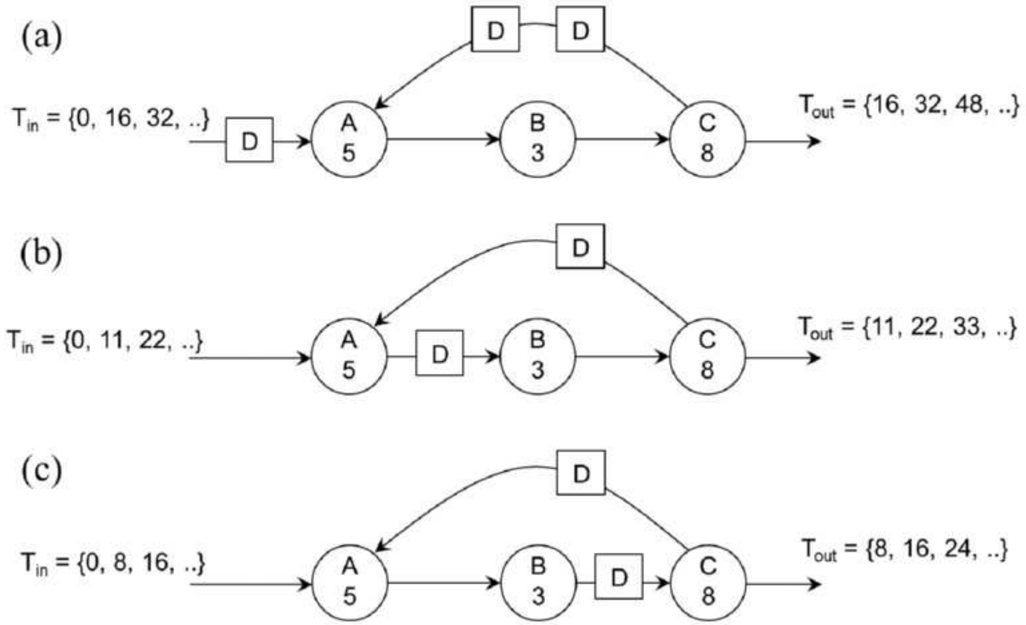
Retiming does **not** introduce new *delay* elements

Evaluation involves inspecting *successive markings* of the DFG and then selecting the one with the best performance


(a)
D
D
T_{in} = {0, 16, 32, ..}
D
A 5
B 3
C 8
T_{out} = {16, 32, 48, ..}
(b)
D
T_{in} = {0, 11, 22, ..}
A 5
D
B 3
C 8
T_{out} = {11, 22, 33, ..}
(c)
D
T_{in} = {0, 8, 16, ..}
A 5
B 3
D
C 8
T_{out} = {8, 16, 24, ..}

(a) has an iteration bound of 8 but produces data on intervals of 16 because of the sequential execution of *actors* A, B and C

Retiming Transformation


(a)
D
D
T_{in} = {0, 16, 32, ..}
D
A 5
B 3
C 8
T_{out} = {16, 32, 48, ..}
(b)
D
T_{in} = {0, 11, 22, ..}
A 5
D
B 3
C 8
T_{out} = {11, 22, 33, ..}
(c)
D
T_{in} = {0, 8, 16, ..}
A 5
B 3
D
C 8
T_{out} = {8, 16, 24, ..}

The next marking (b) is obtained by firing *actor* A, which consumes the *delay* elements on its inputs, and produces a *delay* element at its output

This functionally equivalent configuration *improves* throughput to 1 sample every 11 time units by allowing *actor* A to run in parallel with B and C

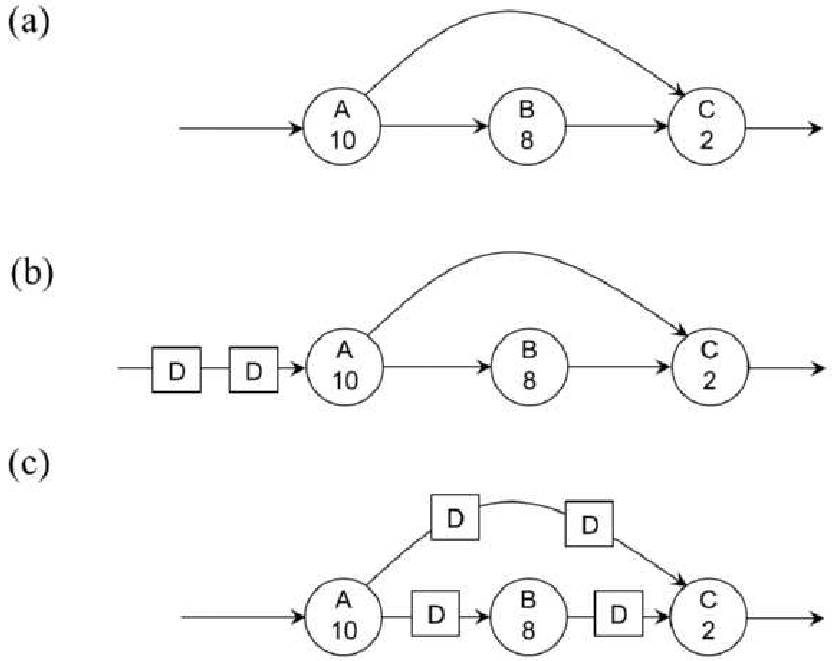
Firing B produces the next marking in (c), which achieves an *iteration bound* of 8 and represents the best that can be obtained

The last marking which fires C creates a configuration nearly equivalent to (a)

Pipelining Transformation

Pipelining increases the throughput at the cost of **increased latency**

Pipelining augments retiming with adding *delay* elements

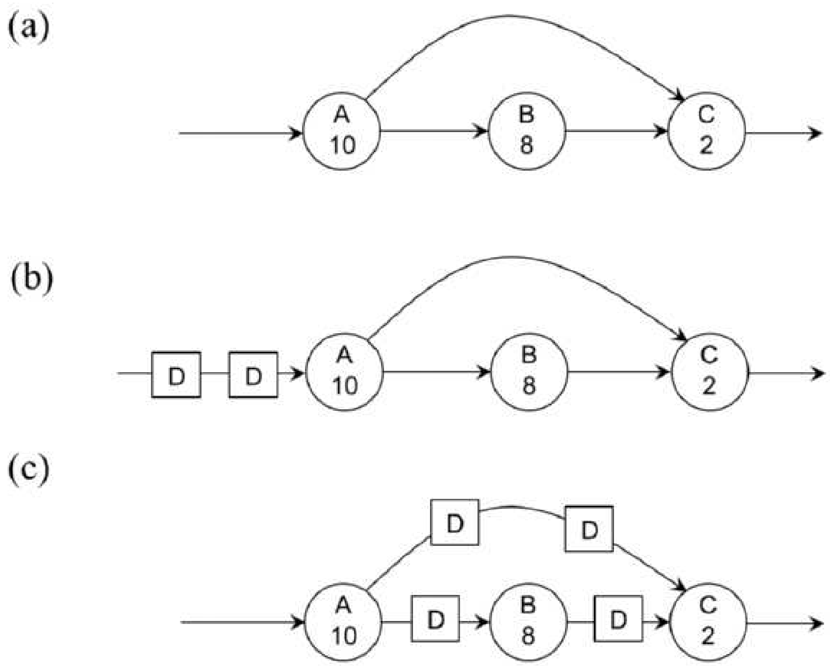


(a) is extended with two pipeline delays in (b)

Adding *delay* elements at the input increases the latency of (a) from 20 to 60

Throughput is 20, i.e., 1 sample every 20 time units

Pipelining Transformation



Retiming of the pipelined graph yields (c) after *firing* A twice and B once, which improves both throughput to 10 and latency to 20

Again we see the slowest pipeline stage determines the best achievable throughput

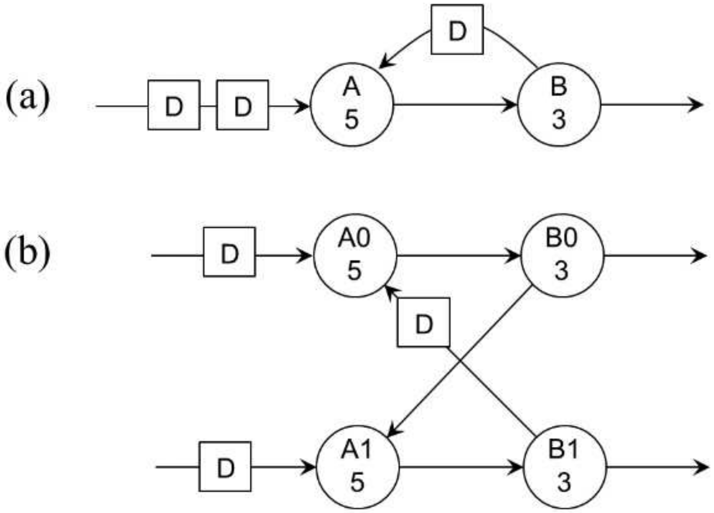
Unfolding Transformation

Unfolding is very similar to the transformation carried out for **multi-rate expansion**

Here, *actor* A in the original DFG is replicated as needed, and interconnections and *delay* elements are redistributed

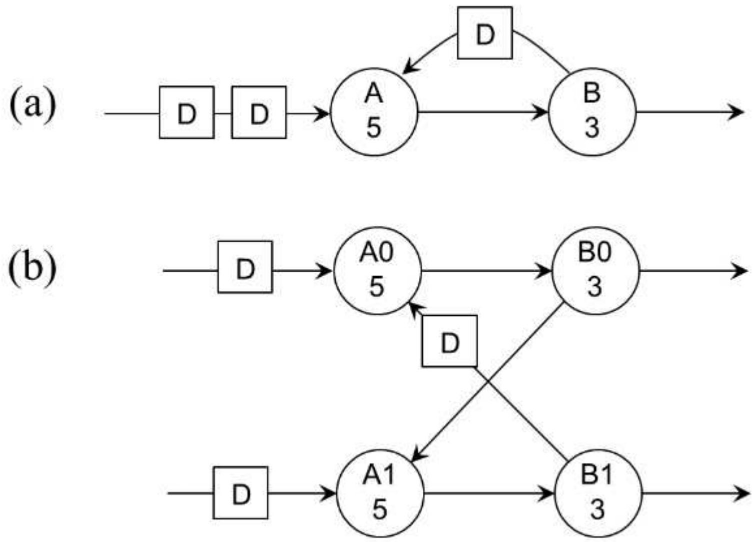
Note the original graph is *single-rate* and goal is to increase sample consumption rate

The text describes the sequence of steps that need to be applied to carry out unfolding



(a) is unfolded two times in (b), showing that the number of inputs and outputs are doubled, allowing twice as much data to be processed per iteration

Unfolding Transformation



Unfolding appears to slow it down, increasing the size of the loop to include A0, B0, A1 and B1 while including only the single *delay* element

Hence, the *iteration bound* of a v-unfolded graph increases v times