Joint Optimization of Data and Program Size in Embedded Software

Praveen K. Murthy

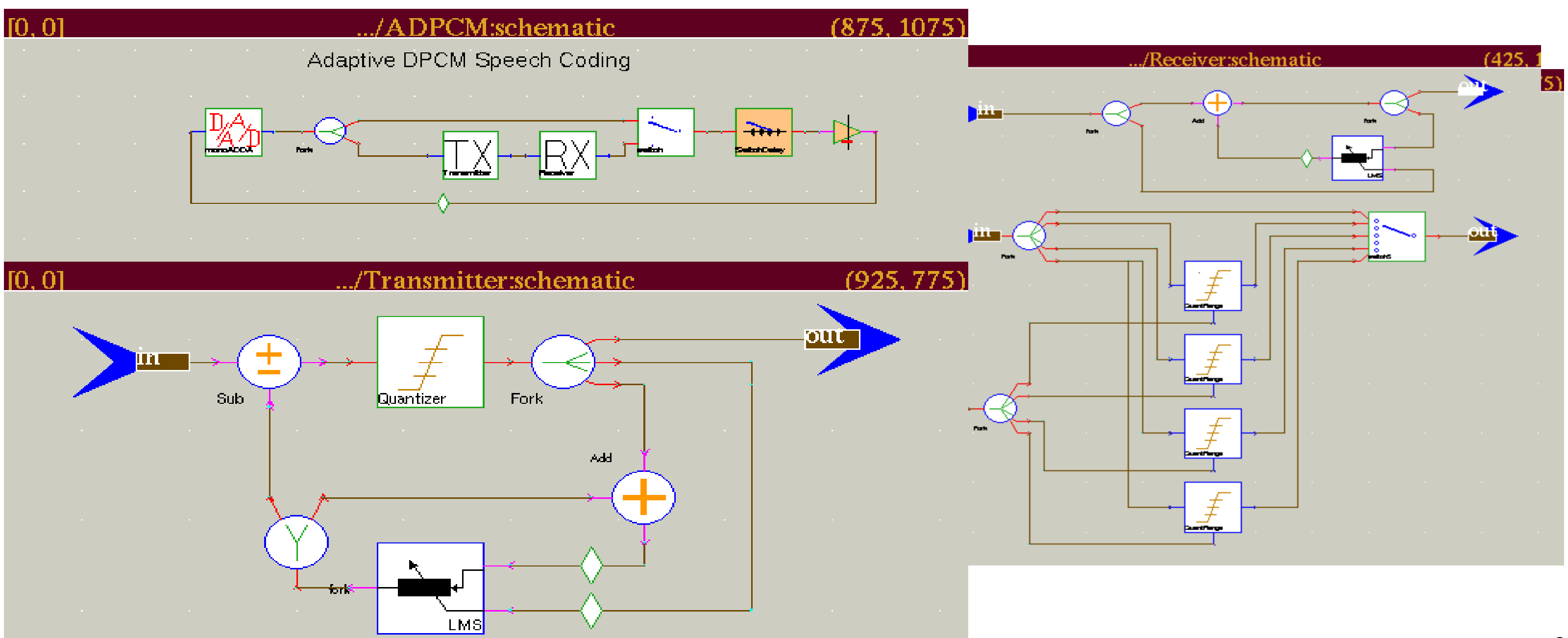
Dept. of EECS,

UC Berkeley

Joint work with:
Shuvra Bhattacharyya (Hitachi),
Edward A. Lee (UC Berkeley)

Block Diagram Languages

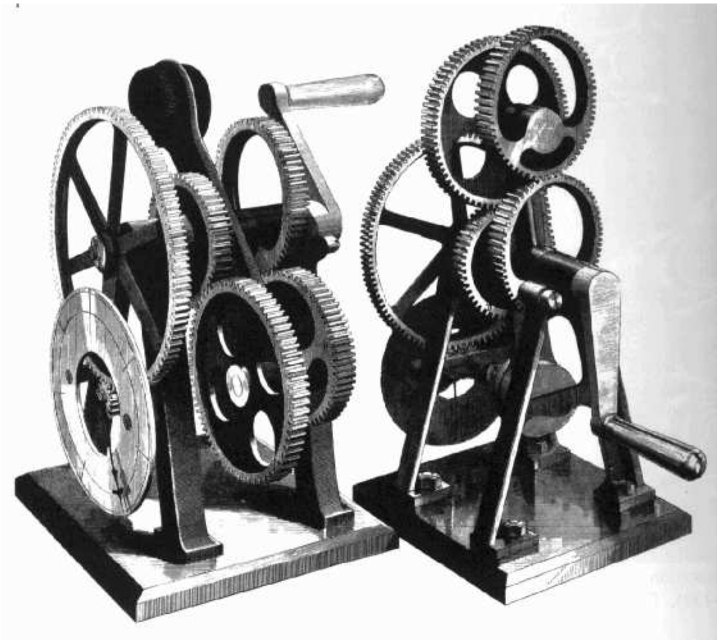
- Appealing visual syntax is natural and intuitive for engineers.
- Encourage software reuse.
- Do not overspecify systems.
- Can be naturally based on well known models of computation that have strong formal properties.



Application designed by members of the Ptolemy project, UC Berkeley

Some models of Computation (MoC)

- Gears (!)
- Synchronous dataflow (SDF)
- Petri Nets
- Boolean dataflow
- Dynamic dataflow
- Kahn process networks
- Discrete events
- Concrete data structures
- Hierarchical finite state machines
- Synchronous/reactive languages
- Hybrid automata
- Communicating sequential processes...



Properties of interest

- Expressiveness
  - Class of programs expressible
  - Turing equivalent = most powerful
- Provable program properties
  - Bounded memory?
  - Freedom from deadlock?
  - ...

In general, more expressive means fewer provable properties, and vice-versa.

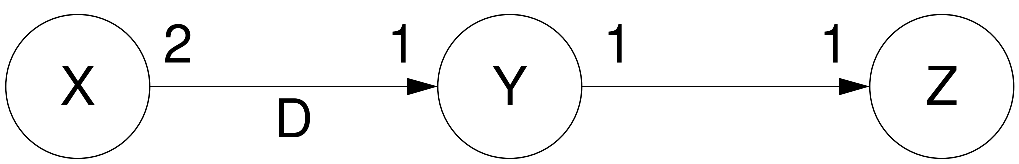
Some MoC properties

- SDF
  - Data-dependent constructs not allowed
  - Ideal for modeling DSP systems since control flow is usually predictable
  - Not really Turing complete
  - Statically schedulable
  - Compile time analysis of deadlock possible
  - Determinate

- Kahn process networks (for example)
  - Turing complete
  - Determinate
  - Bounded memory execution undecidable

Synchronous dataflow

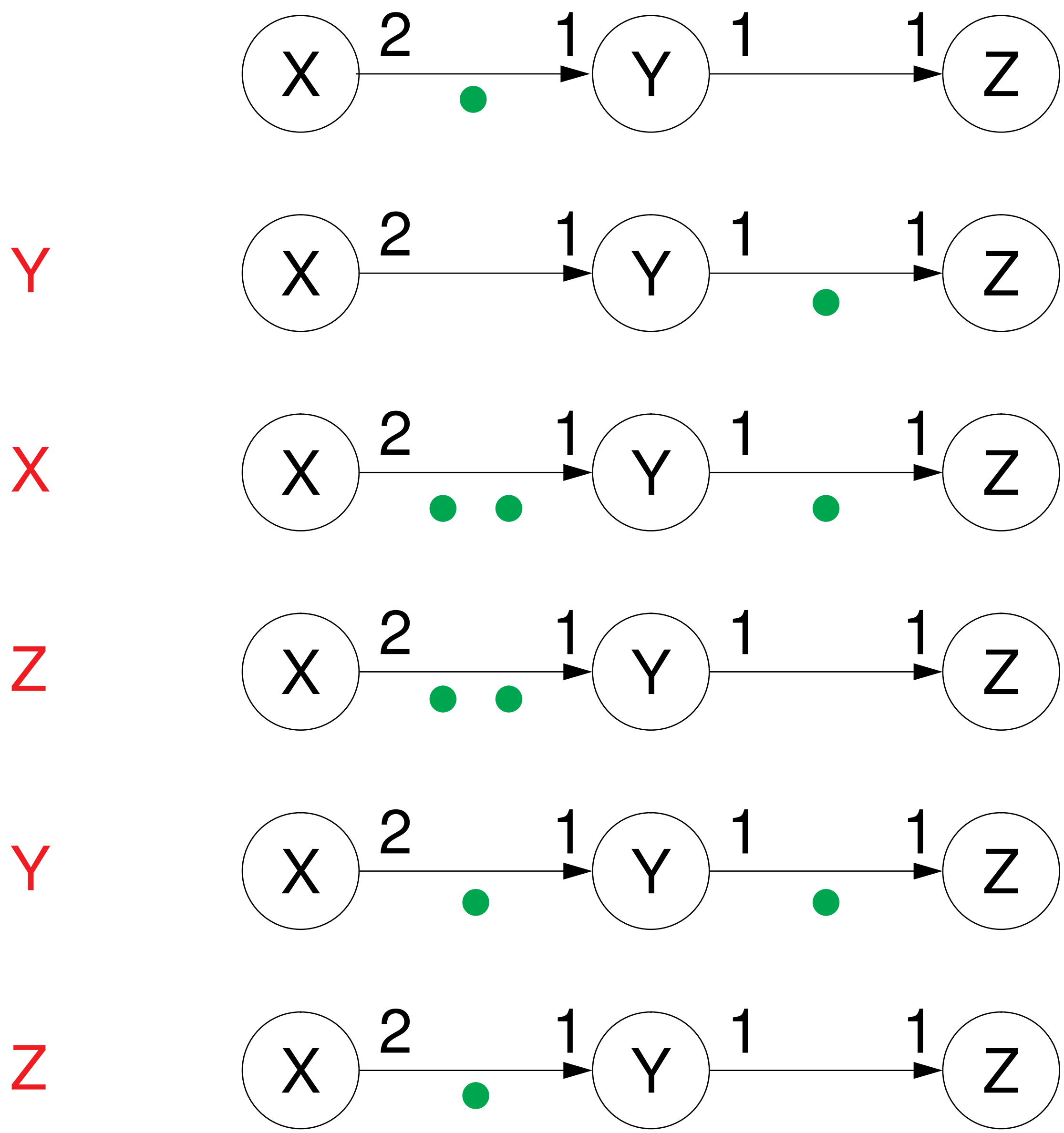
- **The number of tokens produced and consumed by each actor is fixed**
- **Edges may have delays, implemented as initial tokens**
- **Periodic schedules**
- **Unique repetitions vector "q"**



$$q_X = 1, \quad q_Y = 2, \quad q_Z = 2$$

YXZYZ,   XYZYZ,   X(2 YZ)

Periodic schedule example



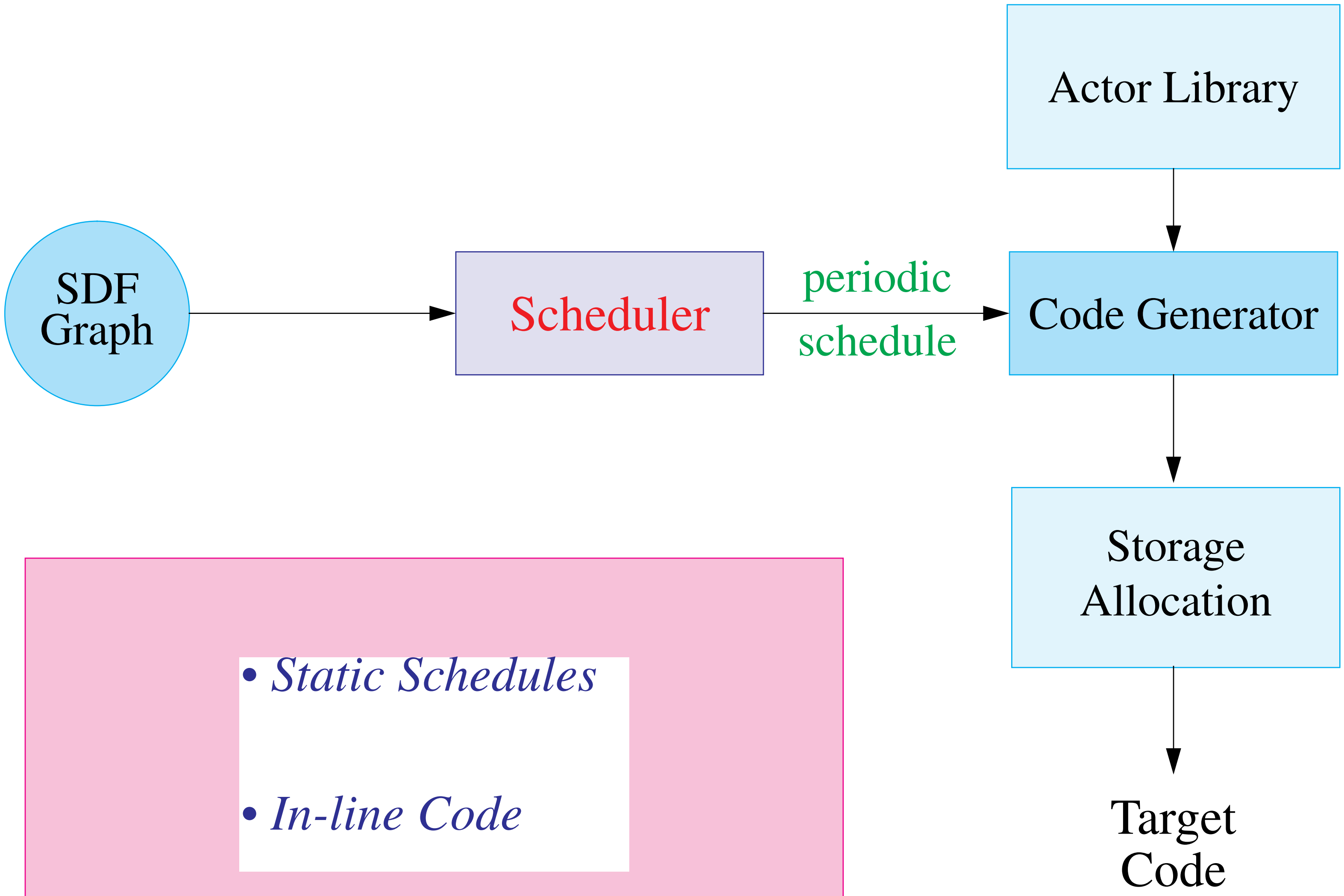
Code generation

- Traditionally, DSPs programmed in assembly language
  - Tedious
  - Error-prone
  - Difficult to maintain and reuse
- Most rapid-prototyping environments provide some sort of code generation capability.

Why not use HLL compilers?

- ~~Imperative languages~~
- Don't do that well [Zivo95]

Software synthesis from synchronous dataflow



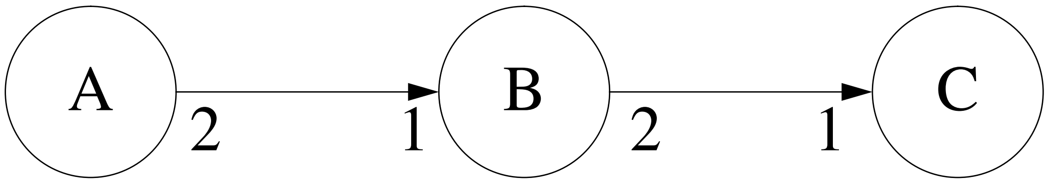
- *Static Schedules*
- *In-line Code*

Advantages and disadvantages

- **Since compiler now has information about the restrictions in overall control flow, can do a better job.**
- **Use of threading does not preclude using traditional compilers on actual actor blocks.**
  - **Actor blocks are usually simple: can even correspond to basic blocks that compilers are good at handling**

- **Code size explosion problem**
- **Buffer sizes explosion problem**

Iteration in SDF graphs



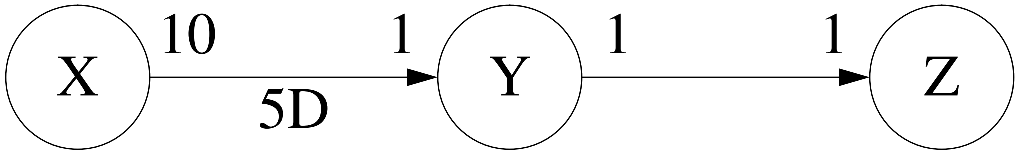
Schedules

A (2 B (2 C)) ← Single Appearance

A (2 B) (4 C) ← Schedules

A B C B CCC

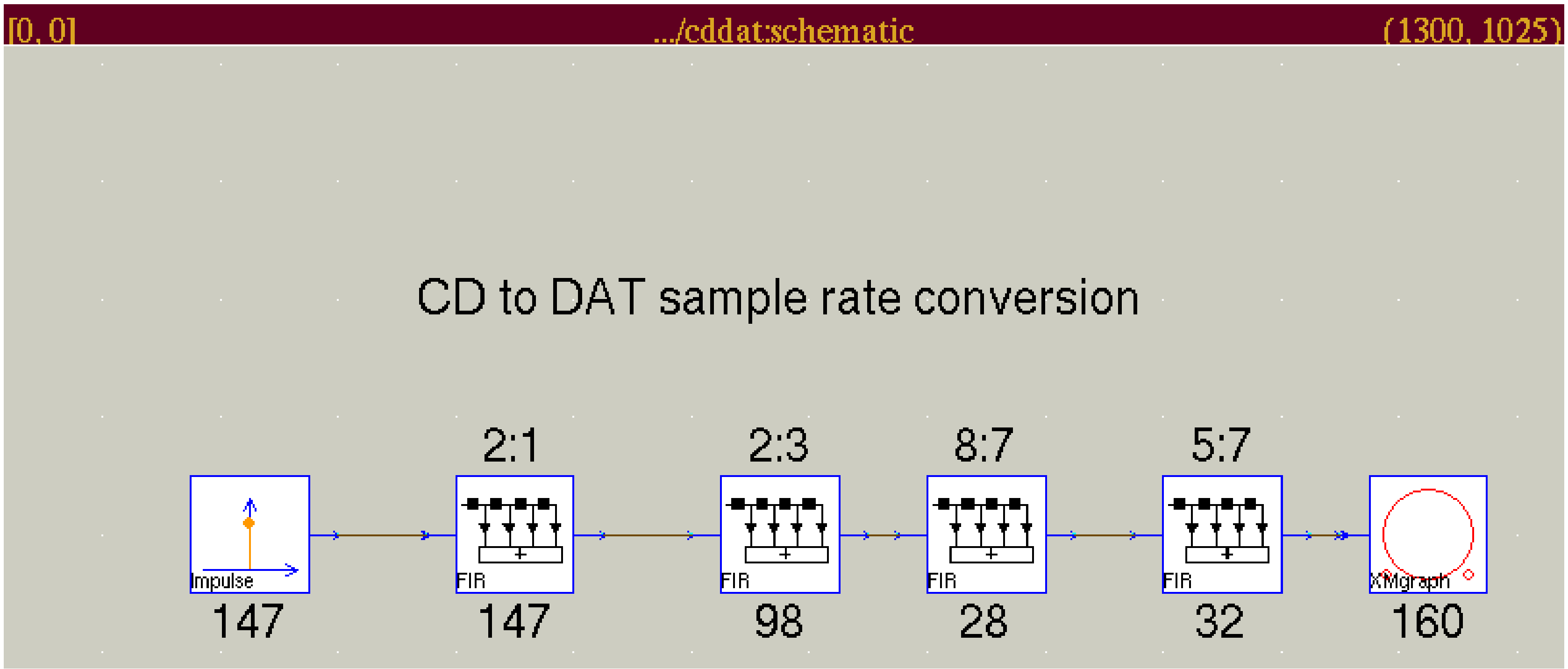
Code size vs. buffer memory



$\mathbf{q}_X = 1, \quad \mathbf{q}_Y = 10, \quad \mathbf{q}_Z = 10$

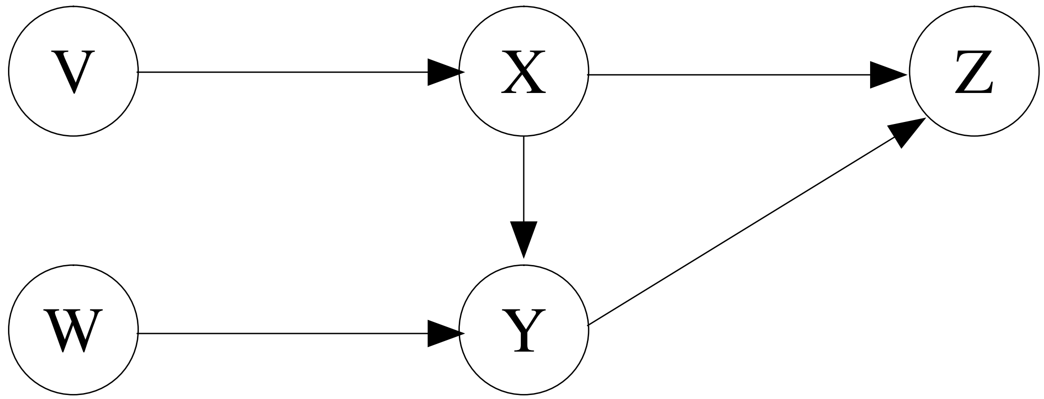
(5 YZ) X (5 YZ)	X (10 Y) (10 Z)	X (10 Y Z)
buffering cost: $10 + 1 = 11$	buffering cost: $15 + 10 = 25$	buffering cost: $15 + 1 = 16$
code size: $S(X) + 2S(Y) + 2S(Z)$	code size: $S(X) + S(Y) + S(Z)$	code size: $S(X) + S(Y) + S(Z)$

Priorities: 1) code size, 2) buffer memory



	Code	Data
Minimum buffer schedule, no looping	**13735**	**32**
Minimum buffer schedule, with looping	**9400**	**32**
Worst minimum code size schedule	**170**	**1021**
Best minimum code size schedule	**170**	**264**

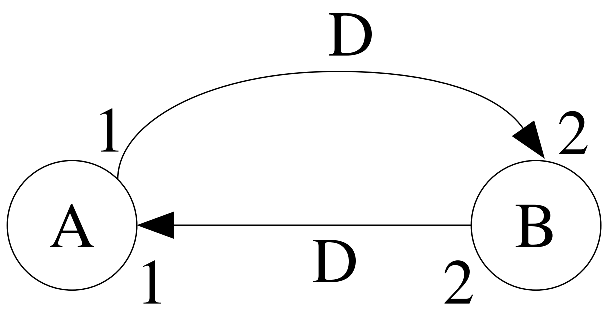
Existence of single appearance schedules



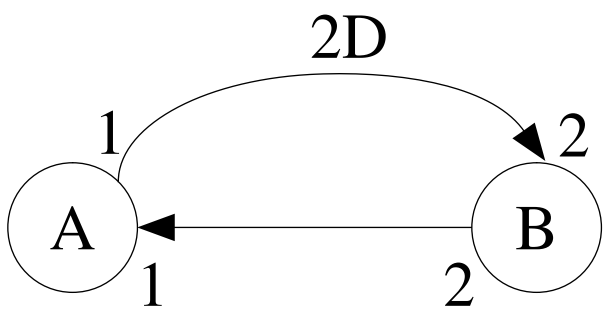
$(q_V\ V)$ $(q_X\ X)$ $(q_W\ W)$ $(q_Y\ Y)$ $(q_Z\ Z)$

Acyclic graphs always have single appearance schedules

Existence of single appearance schedules



A B A

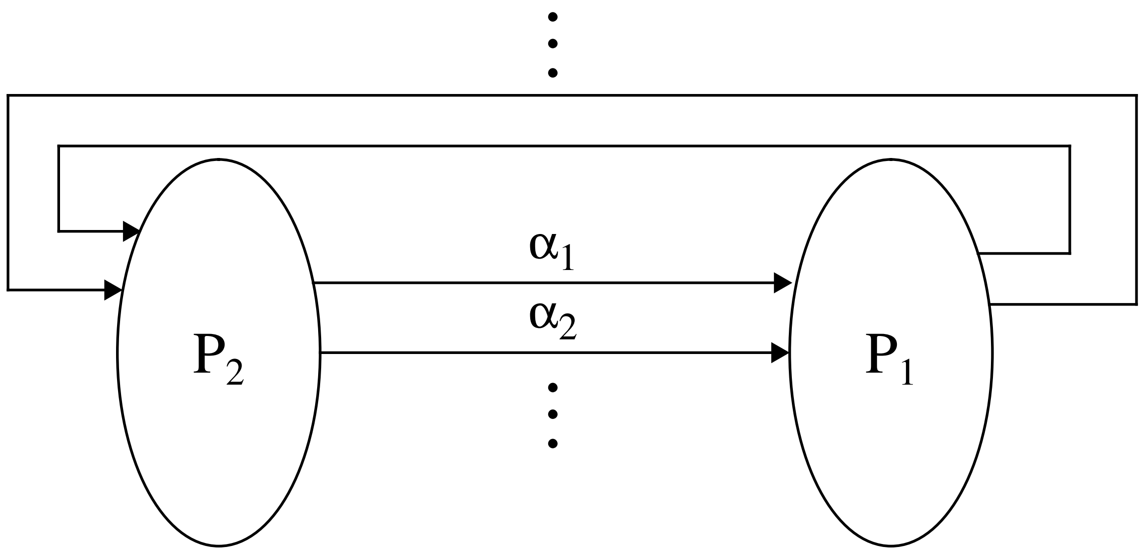


B (2 A)

Graphs that contain cycles may or may not have single appearance schedules, depending on the delays.

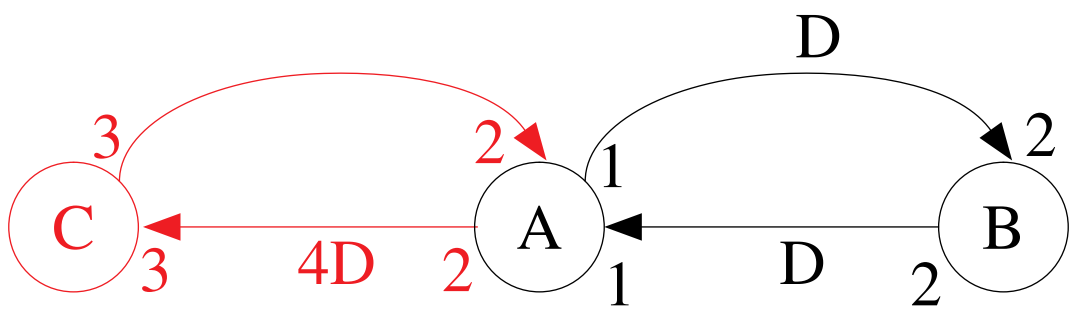
Subindependence

- A strongly connected SDF graph has a single appearance schedule only if its actors can be partitioned into P_1 and P_2 such that P_1 does not depend on data produced by P_2 in the same schedule period.



- *delay*$(\alpha_i) \geq$ (number of tokens consumed from α_i each schedule period)

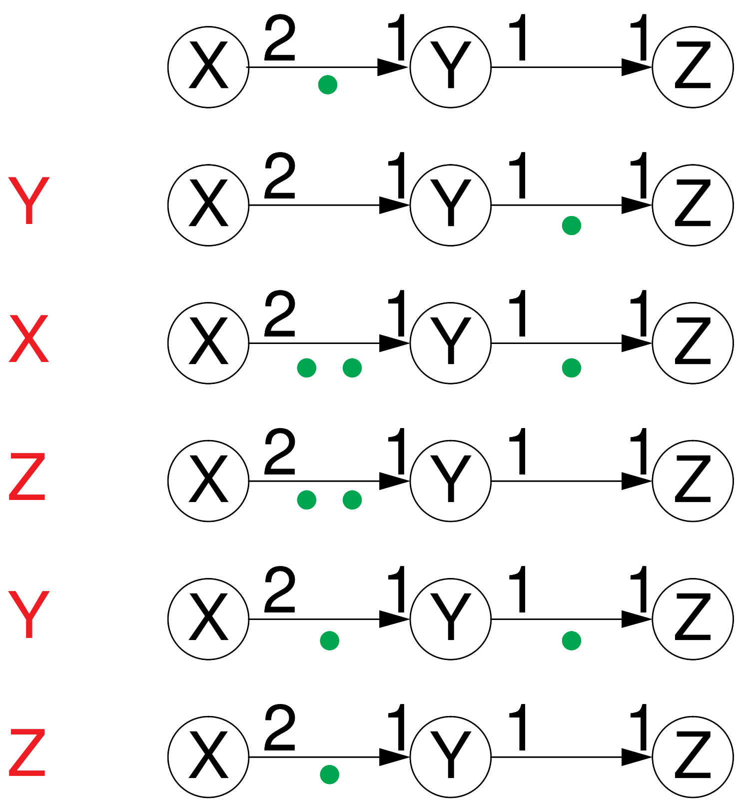
Tight interdependence



- **A strongly connected SDF graph is tightly interdependent if it does not have a subindependent partition.**
- **If Z_1 and Z_2 are subsets of nodes in an SDF graph such that $Z_1 \cap Z_2 \neq \emptyset$ and *subgraph*(Z_1) & *subgraph*(Z_2) are both tightly interdependent, then *subgraph*($Z_1 \cup Z_2$) is tightly interdependent.**
- **Tightly interdependent components.**

Buffering model

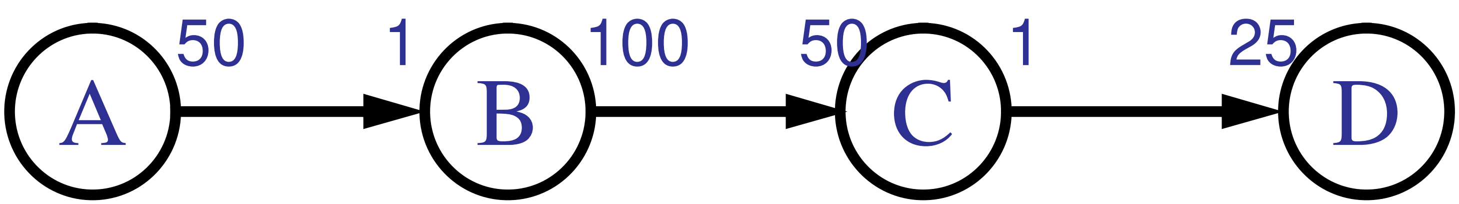
- Buffer on every arc in the graph.
- Size of a buffer is given by the maximum number of tokens queued on the arc in the schedule.
- Total buffering cost is sum of individual buffer sizes.



buffering cost = 2 + 1 = 3

Advantages over alternative buffering models

- **Alternative #1: *Flat* single appearance schedules with shared buffers**
  - Buffering requirement can be very large for some graphs
  - Does not handle delays well
  - Latency is maximized
- **Alternative #2: Use nested schedules with buffer sharing**
  - More awkward to implement
  - Cost function is more complicated



A (50 B) (100 C) (4 D): Cost = 5000

A (2 (25 (B (2 C))) (2 D)): Cost = 200

Minimum buffer single appearance schedules

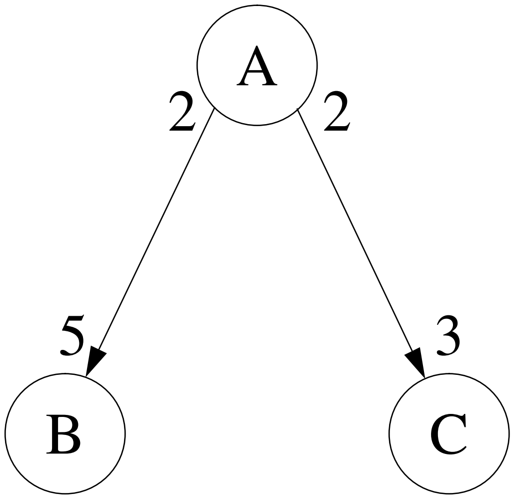
Schedule	Buffering Cost
(9 A)(12 B)(12 C)(8 D)	36 + 12 + 24 = **72**
(3 (3 A)(4 B C))(8 D)	12 + 1 + 24 = **37**
(3 (3 A)(4 B))(4 (3 C)(2 D))	12 + 12 + 6 = **30**

Complexity

- **Elements of difficulty**
  - — Cyclic paths (feedback)
  - — Delays
  - — Iteration (mismatches in production and consumption parameters)
- **NP-complete subproblems**
  - — Cycles allowed, “minimal” delays, no iteration (Reduce feedback arc set)
  - — Arbitrary delays, acyclic, no iteration (Reduce vertex cover)
- **Open problem: complexity of scheduling acyclic, delayless graphs that may have arbitrary iteration**

Minimum buffer single appearance schedules

single appearance schedule $\Leftrightarrow$ a parenthesization of a lexical ordering



- (3 (5 A) (2 B)) (10 C) buff. cost = 40
- (15 A) (2 (3 B) (5 C)) buff. cost = 60
- (15 A) (6 B) (10 C) buff. cost = 60
- (5 (3 A) (2 C)) (6 B) buff. cost = 36
- (15 A) (2 (5 C) (3 B)) buff. cost = 60

$\mathbf{q} = (15, 6, 10)$

Dynamic programming post optimization

- Given a lexical ordering, computes an optimal parenthesization.
- Time complexity $O(n^3)$.

$$\ldots((x_i x_{i+1} \ldots x_k)(x_{k+1} \ldots x_{j-1} x_j))\ldots$$

left subchain cost

split cost

right subchain cost

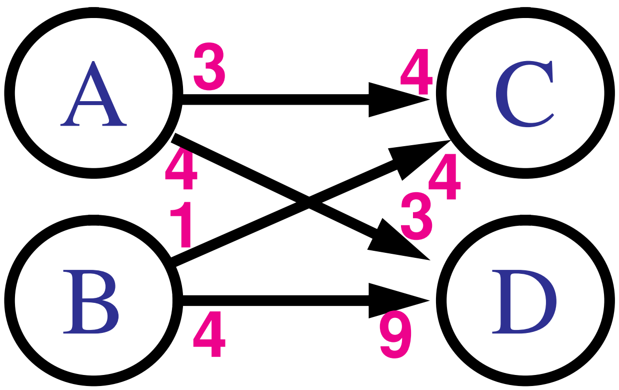
$$b[i, j] = MIN_{i \le k < j}\{b[i, k] + b[k+1, j] + c_{ij}[k]\}$$

where $c_{ij}[k] = \dfrac{r_k O_k}{gcd(r_i, \ldots, r_j)}$

- If graph is well-ordered, then this is optimal.

General Acyclic Graphs

- Any *topological sort* of an acyclic graph leads to a set of valid single appearance schedules.
- An acyclic graph can have an exponential number of topological sorts in general: a complete $2n$ node bipartite graph has $(n!)^2$ topological sorts.
- The problem is to pick the topological sort that leads to the best nested schedule when nested optimally using dynamic programming algorithm.



(3 (4 **A**) (3 (4 **B**) **C**)) (16 **D**): 208

(4 (3 **A**) (9 **B**) (4 **D**)) (9 **C**): 120

Heuristics for generating lexical orderings

- **We give two heuristic techniques for finding buffer-optimal schedules for acyclic graphs:**

  - **First technique is a *top-down* approach using min-cuts called *Recursive Partitioning by Minimum Cuts* (RPMC).**

  — Effective for irregular topologies

  - **Second technique is a *bottom-up* approach using clustering called *Acyclic Pairwise Grouping of Adjacent Nodes* (APGAN).**

  — Effective for regular topologies
  — Optimal for a class of graphs

Complementary:

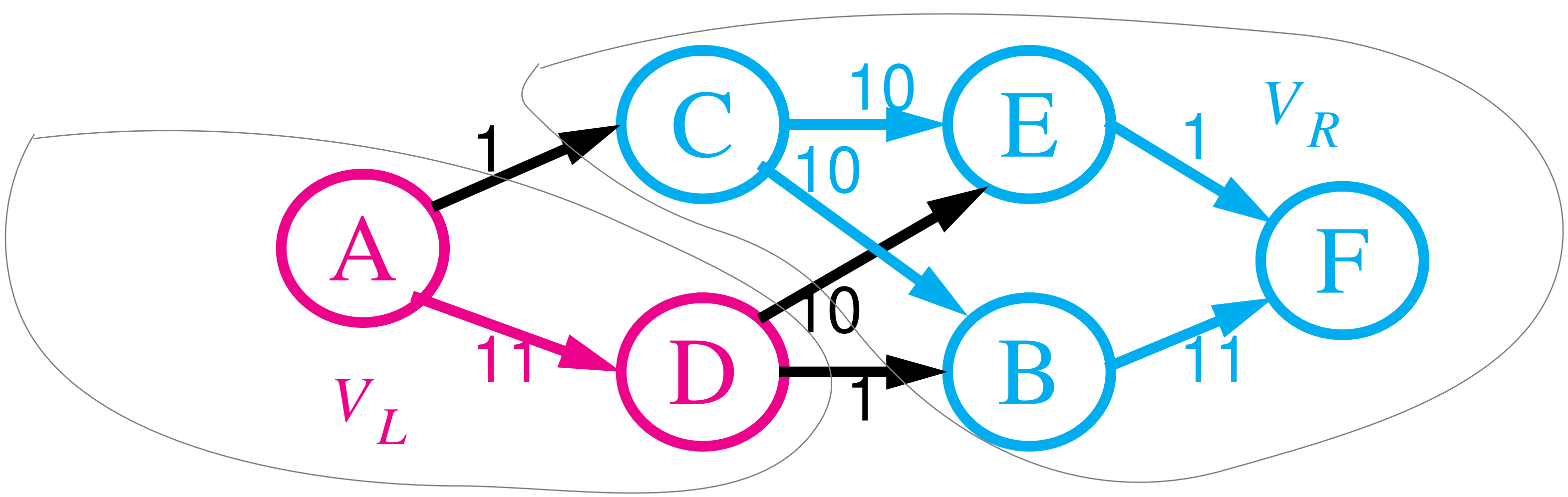
often, when one does poorly, the other does well

Recursive Partitioning by Min Cuts

Idea: Find a *cut* of the graph such that

- **a) All arcs cross the cut in the forward direction.**
- **b) The cut results in fairly even-sized sets.**
- **c) Amount of data crossing the cut is minimized.**

Recursively schedule the nodes on the left side of the cut before nodes on the right side of the cut.



RPMC (cont'd)

- **Splitting the graph where the minimum amount of data is transferred is a *greedy* approach and is not optimal in general.**

- **Finding the minimum cut such that all of the conditions a,b, and c are satisfied is itself a difficult problem:**

  — Methods based on max-flow-min-cut theorem do not work.
  — Graph partitioning when the size of the partition has to be bounded is NP-complete.

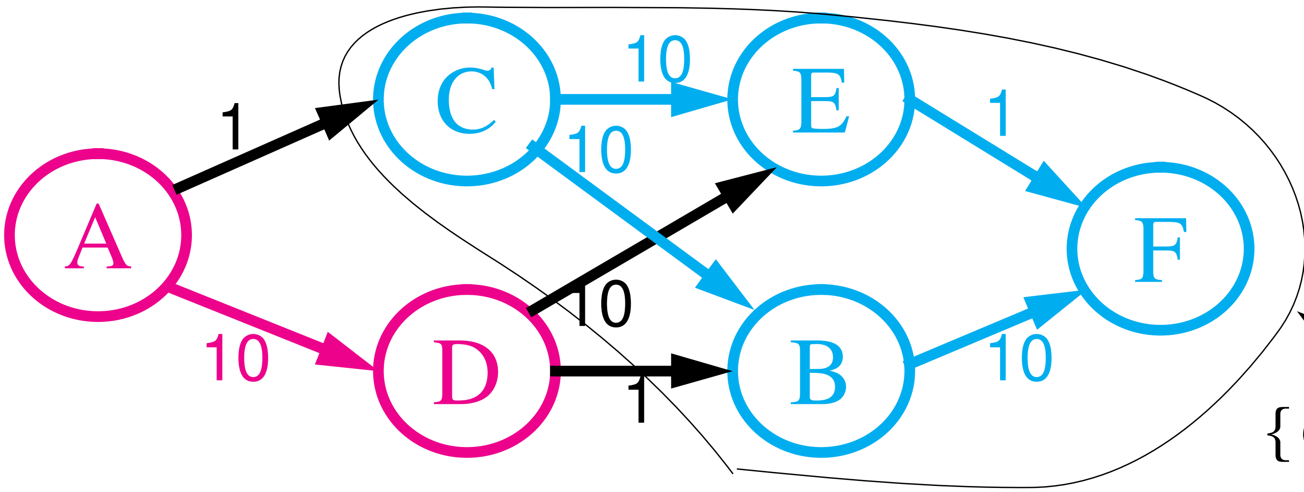
- **Therefore, a heuristic solution is needed.**

A Heuristic for Legal Min Cuts

- Let $V_R(u)$ be the set of nodes consisting of u and its descendents. Let $V_L = V \setminus V_R(u)$.
- This forms a cut satisfying condition (a).
- Perform a local optimization by moving those nodes from V_L that reduce the cost into $V_R(u)$.
- Do this for all nodes u in the graph.

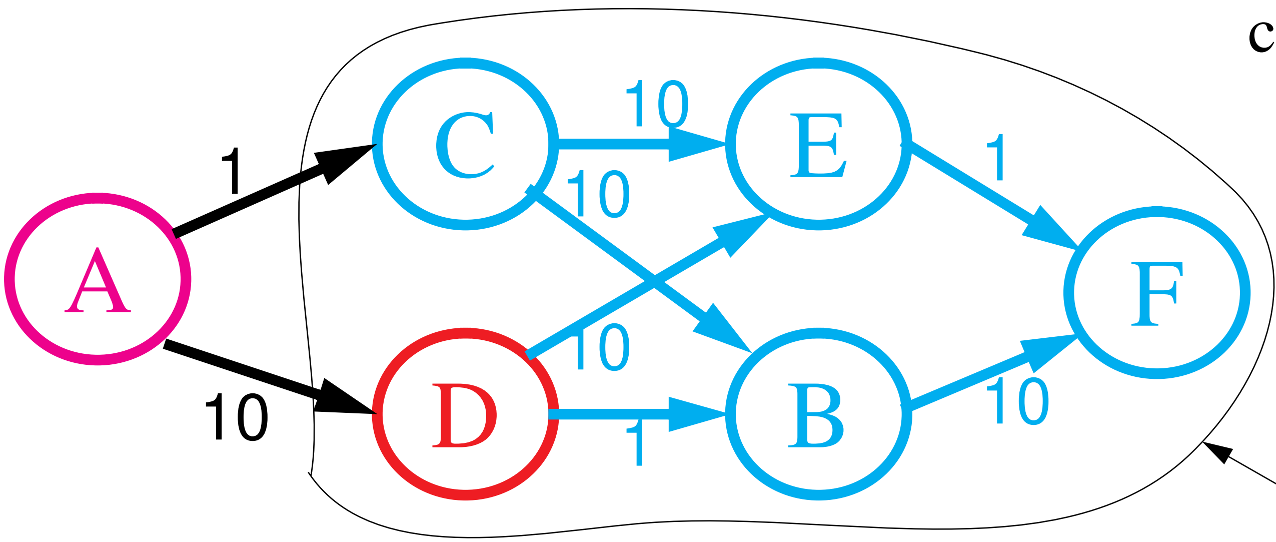
- Repeat above steps to generate cuts obtained by letting $V_L(u)$ be the set of nodes consisting of u and it ancestors, and letting $V_R = V \setminus V_L(u)$.
- Keep the cut with the lowest cost.
- Runs in time $O(|V||E| + |V|^2 \bullet \log(|V|))$.

Example



$\{C\} \cup desc(C)$,

$cost = 12$



$\{C\} \cup desc(C) \cup \{D\}$,

$cost = 11$

RPMC Algorithm

- **Find heuristic minimum cut of the graph into sets V_L and V_R.**
  - **The top level schedule is given by**

$$S(V) = (q_L S(V_L))(q_R S(V_R)) \quad \textbf{where} \quad q_i = gcd\{r(v) : v \in V_i\}, i = L, R \text{ .}$$

- **Continue recursively until all nodes have been scheduled.**

- **Post-process resulting schedule by recomputing an optimum nesting of the loops using dynamic programming algorithm with the lexical ordering generated by RPMC.**

- **Runs in time $O(|V|^3)$ for sparse graphs.**

Acyclic Pairwise Grouping of Nodes (APGAN)

Idea: Develop a loop hierarchy by clustering two adjacent nodes at each step.

Definition: *Clustering* means combining two or more nodes into one hierarchical node.

- The graph with the hierarchical node instead of the nodes that were clustered is called the *clustered graph*.

Definition: A *clusterizable* pair of nodes is a pair of nodes that, when clustered, does not cause *deadlock*.

- A sufficient condition for clusterizability: Two nodes are clusterizable if clustering them does not introduce a cycle in the clustered graph.

APGAN Algorithm

- **Cluster two nodes that maximize $gcd\{r(A), r(B)\}$ over all clusterizable pairs $\{A, B\}$.**

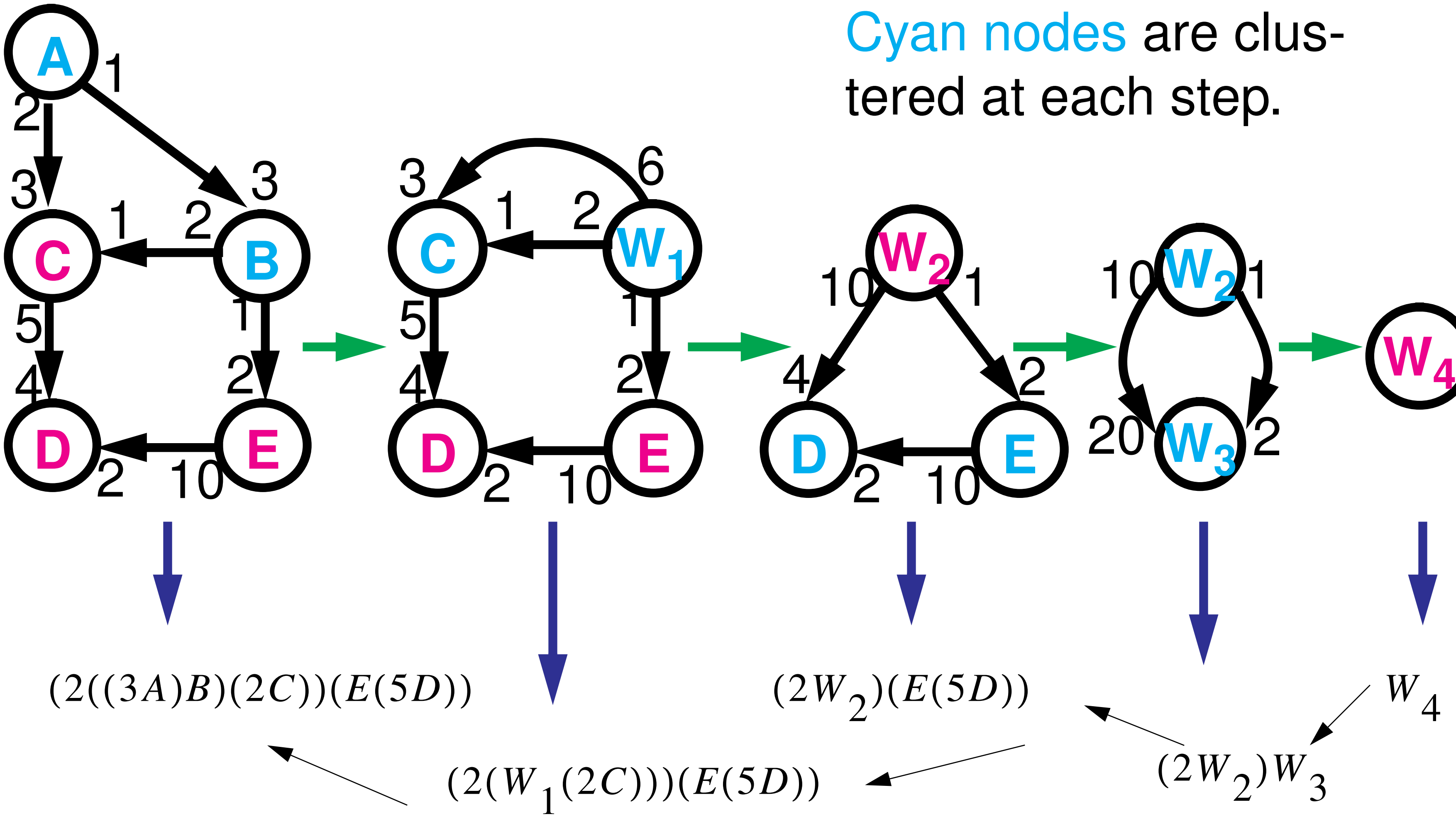
- **Continue until only one node is left in the clustered graph**
  - **This is similar to the Huffman coding algorithm.**

- **After constructing cluster hierarchy, retrace steps to determine the nested schedule.**

- **Post-process the schedule using dynamic programming to generate an optimal nesting for the lexical ordering generated by APGAN.**
- **Runs in time $O(|V|^3)$ for sparse graphs.**

APGAN in action

Cyan nodes are clustered at each step.



Optimality of PGAN

- **Definition:** The ***buffer memory lower bound*** for a (delayless) arc (e) is given by

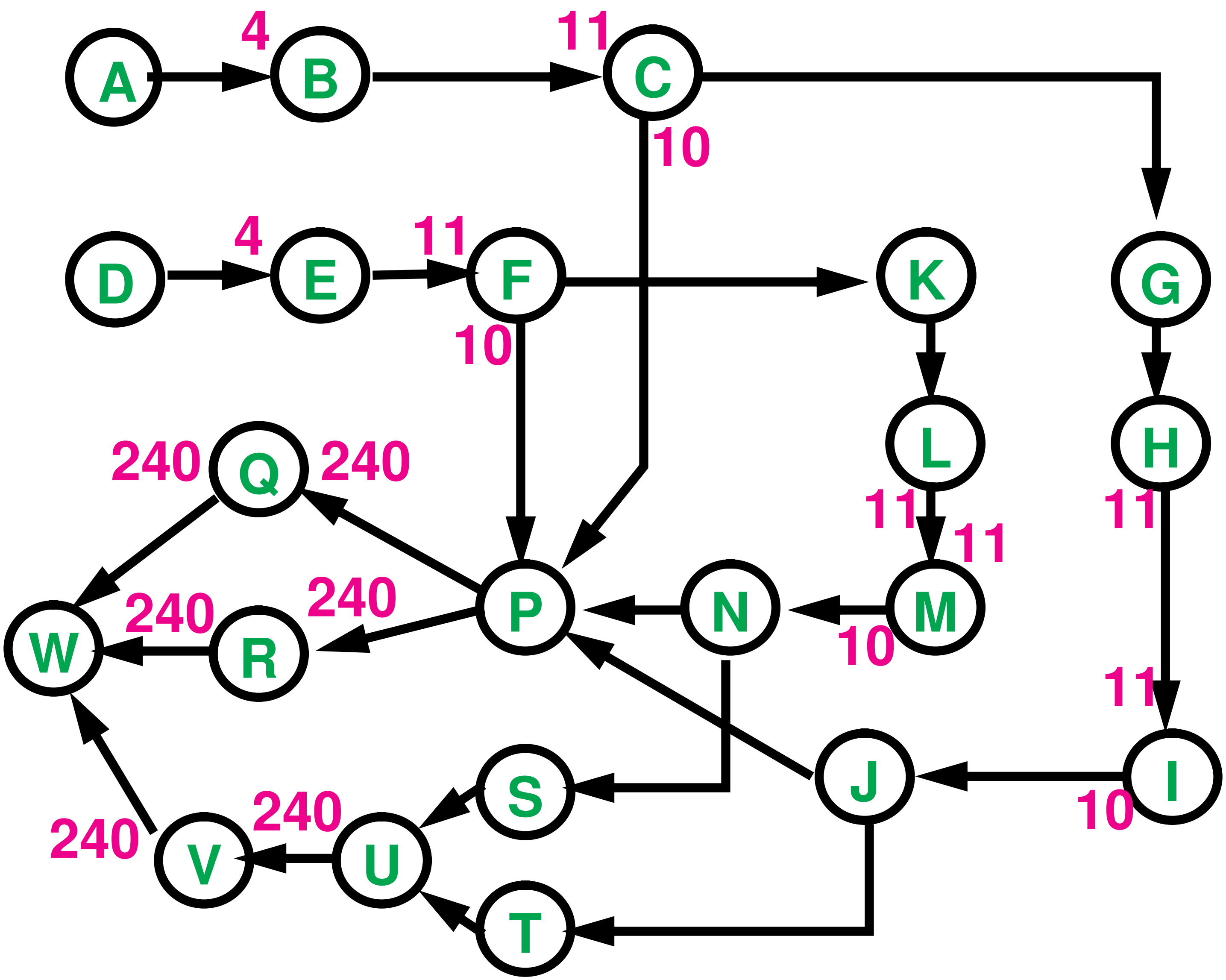
$$BMLB(u, v) = \frac{cons(e)prod(e)}{gcd\{cons(e), prod(e)\}}$$

— **This represents the least amount of buffering needed on this arc in any single appearance schedule.**

- **Definition:** A ***BMLB schedule*** for an acyclic SDF graph is a single appearance schedule whose buffering cost is equal to the sum of the BMLB costs for each arc.

- **Theorem:** The PGAN algorithm will find a BMLB schedule whenever one exists if $delay(e) < \tau(e)$ for each edge e.

Example: mobile satellite receiver



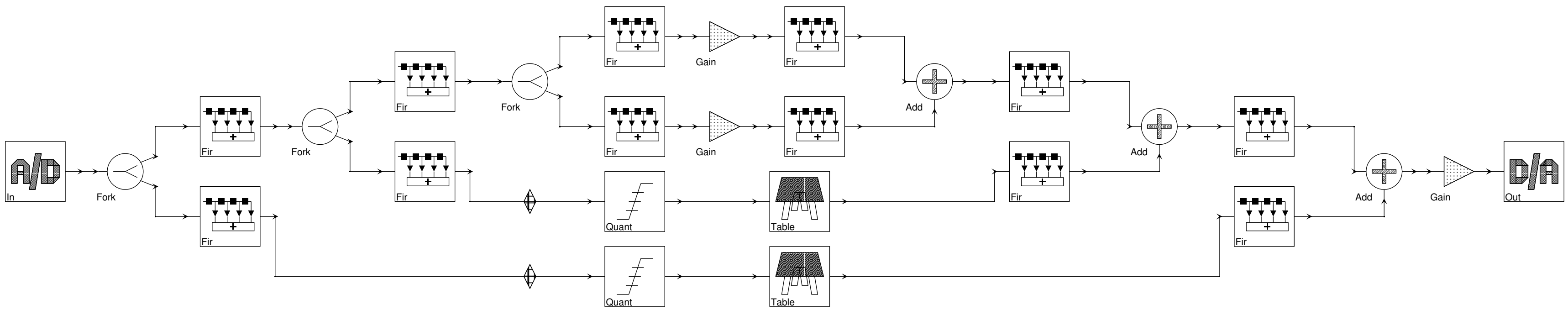
BMLB = 1540

PGAN = 1540

RPMC = 2480

Nonuniform filter bank

A Nonuniform filterbank.
The highpass component retains 1/3 of the spectrum at each stage while the lowpass retains 2/3 of the spectrum



Analysis sections

Synthesis sections

BMLB = 85

RPMC = 128

PGAN = 137

Performance on practical examples

Performance of the two heuristics on various acyclic graphs.

System	BMUB	BMLB	PGAN	RPMC	Average Random	Graph size(nodes/ arcs)
Fractional decimation	61	47	47	52	52	26/30
Laplacian pyramid	115	95	99	99	102	12/13
Nonuniform filterbank (1/3,2/3 splits, 4 channels)	466	85	137	128	172	27/29
Nonuniform filterbank (1/3,2/3 splits, 6 channels)	4853	224	756	589	1025	43/47
QMF nonuniform-tree filterbank	284	154	160	171	177	42/45
QMF filterbank (one-sided tree)	162	102	108	110	112	20/22
QMF analysis only	248	35	35	35	43	26/25
QMF Tree filterbank (4 channels)	84	46	46	55	53	32/34
QMF Tree filterbank (8 channels)	152	78	78	87	93	44/50
QMF Tree filterbank (16 channels)	400	166	166	200	227	92/106

Performance on random graphs

RPMC < PGAN	63%
PGAN < RPMC	37%
RPMC < min(2 random)	83%
PGAN < min(2 random)	68%
RPMC < min(4 random)	75%
PGAN < min(4 random)	61%
min(RPMC,PGAN) < min(4 random)	87%
RPMC < PGAN by more than 10%	45%
RPMC < PGAN by more than 20%	35%
PGAN < RPMC by more than 10%	23%
PGAN < RPMC by more than 20%	14%

Summary

Goal: to map dataflow specifications of DSP systems into efficient implementations on embedded processors

- **Code size minimization**
  - — Single appearance schedules
  - — Tightly interdependent components
  - — Scheduling framework
- **Buffer memory minimization**
  - — Parenthesizations of lexical orderings
  - — Problem for even acyclic, HSDF graphs is NP-complete
  - — PGAN and RPMC: complementary heuristics
  - — BMLB schedules
- **Implementations**
  - — Ptolemy
  - — SPW (Cadence)