Dataflow model of computation and dataflow execution

Luca Benini

DEIS Università di Bologna

2

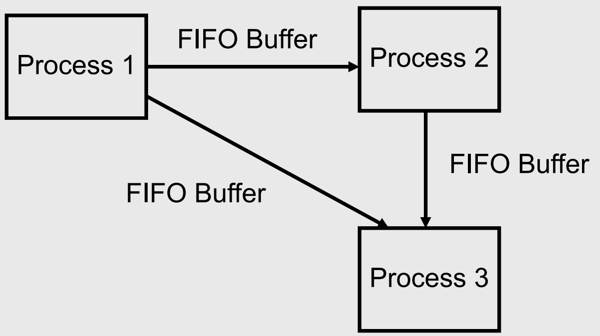
Philosophy of Dataflow

- Drastically different way of looking at computation
- Von Neumann imperative language style: program counter is king
- Dataflow language: movement of data the priority
- Scheduling responsibility of the system, not the programmer

3

Dataflow Model of Computation

- Processes communicating through FIFO buffers



4

Dataflow Semantics

- Every process runs simultaneously
- Processes can be described with imperative code
- Compute … compute … receive … compute … transmit
- Processes can *only* communicate through buffers

5

Dataflow Communication

- Communication is *only* through buffers
- Buffers usually treated as unbounded for flexibility
- Sequence of tokens read guaranteed to be the same as the sequence of tokens written
- Destructive read: reading a value from a buffer removes the value
- Much more predictable than shared memory

6

Applications of Dataflow

- Not a good fit for, say, a word processor
- Good for signal-processing applications
- Anything that deals with a continuous stream of data

- Becomes easy to parallelize
- Buffers typically used for signal processing applications anyway

7

Kahn Process Networks

- Proposed by Kahn in 1974 as a general-purpose scheme for parallel programming
- Laid the theoretical foundation for dataflow
- Unique attribute: deterministic
- Difficult to schedule
- Too flexible to make efficient, not flexible enough for a wide class of applications
- Never put to widespread use

8

Kahn Process Networks

- Key idea:

Reading an empty channel blocks until data is available

- No other mechanism for sampling communication channel's contents
  - Can't check to see whether buffer is empty
  - Can't wait on multiple channels at once

9

Kahn Processes

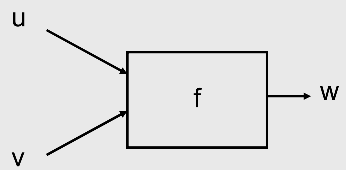
- A C-like function (Kahn used Algol)
- Arguments include FIFO channels
- Language augmented with send() and wait() operations that write and read from channels

10

A Kahn Process

- From Kahn's original 1974 paper

```
process f(in int u, in int v, out int w)
{
  int i; bool b = true;
  for (;;) {
    i = b ? wait(u) : wait(w);
    printf("%i\n", i);
    send(i, w);
    b = !b;
  }
}
```



Process alternately reads from u and v, prints the data value, and writes it to w

11

A Kahn Process

- From Kahn's original 1974 paper

```
process f(in int u, in int v, out int w)
{
  int i; bool b = true;
  for (;;) {
    i = b ? wait(u) : wait(w);
    printf("%i\n", i);
    send(i, w);
    b = !b;
  }
}
```

Process interface includes FIFOs

wait() returns the next token in an input FIFO, blocking if it's empty

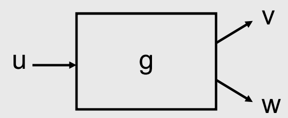
send() writes a data value on an output FIFO

12

A Kahn Process

- From Kahn's original 1974 paper

```
process g(in int u, out int v, out int w)
{
  int i; bool b = true;
  for(;;) {
    i = wait(u);
    if (b) send(i, v); else send(i, w);
    b = !b;
  }
}
```



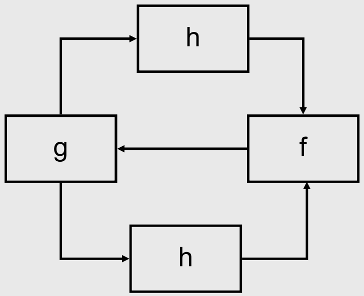
Process reads from u and alternately copies it to v and w

13

A Kahn System

- Prints an alternating sequence of 0's and 1's

Emits a 1 then copies input to output



Emits a 0 then copies input to output

14

Proof of Determinism

- Because a process can't check the contents of buffers, only read from them, each process only sees sequence of data values coming in on buffers

- Behavior of process:

Compute … read … compute … write … read … compute

- Values written only depend on program state
- Computation only depends on program state
- Reads always return sequence of data values, nothing more

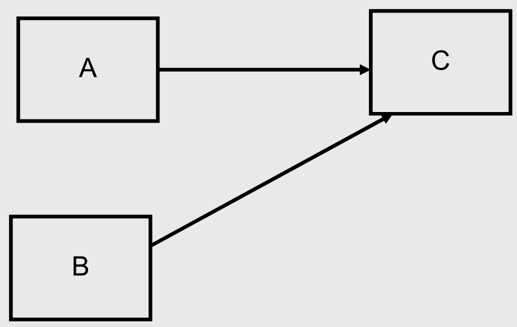
15 **Determinism**

- Another way to see it:

- If I'm a process, I am only affected by the sequence of tokens on my inputs
- I can't tell whether they arrive early, late, or in what order
- I will behave the same in any case
- Thus, the sequence of tokens I put on my outputs is the same regardless of the timing of the tokens on my inputs

16 **Scheduling Kahn Networks**

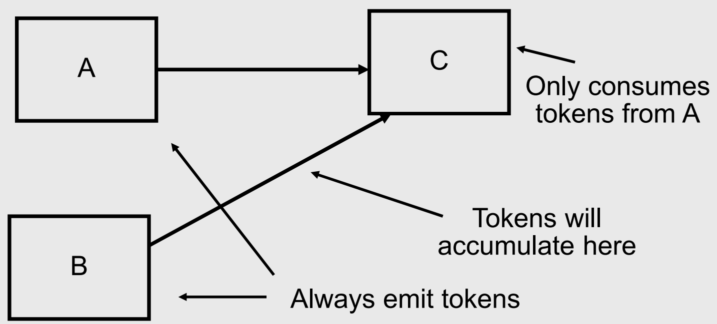
- Challenge is running processes without accumulating tokens



17

Scheduling Kahn Networks

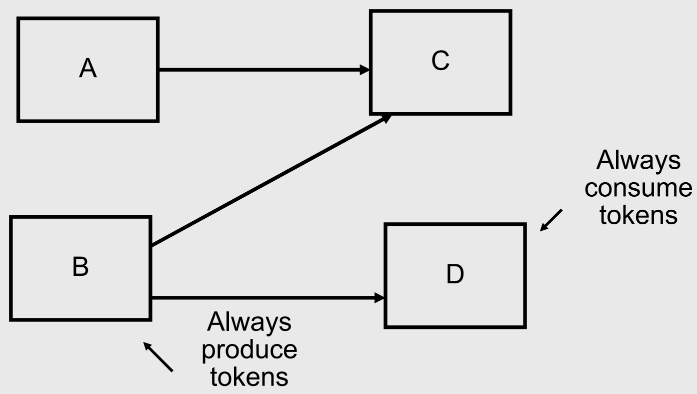
- Challenge is running processes without accumulating tokens



18

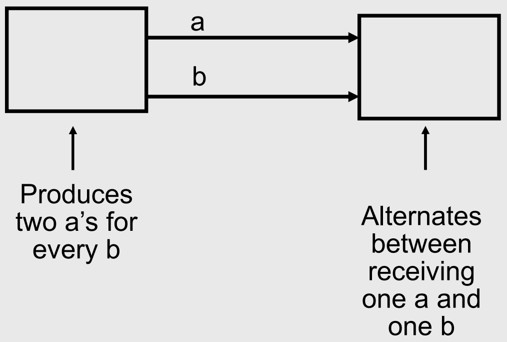
Demand-driven Scheduling?

- Apparent solution: only run a process whose outputs are being actively solicited
- However...



19 Other Difficult Systems

- Not all systems can be scheduled without token accumulation



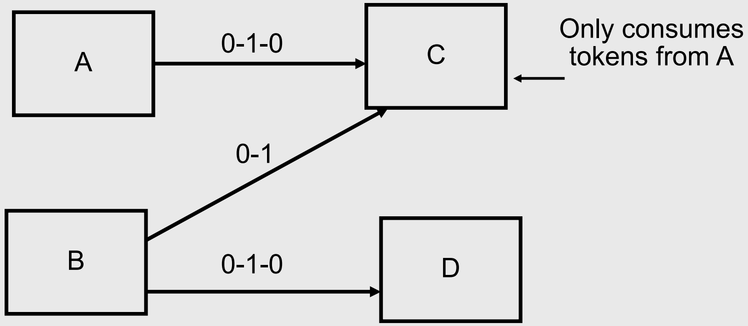
20 Tom Parks' Algorithm

- Schedules a Kahn Process Network in bounded memory if it is possible
- Start with bounded buffers
- Use any scheduling technique that avoids buffer overflow
- If system deadlocks because of buffer overflow, increase size of smallest buffer and continue

21

Parks' Algorithm in Action

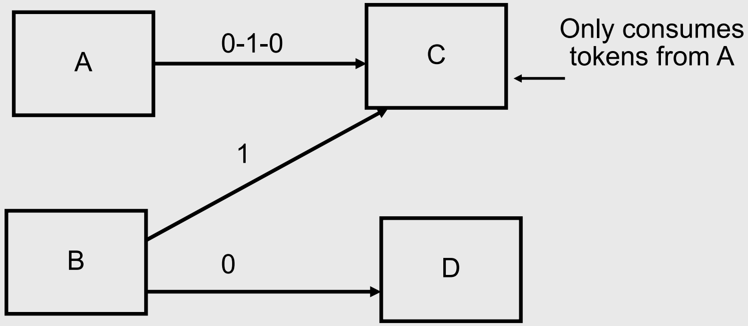
- Start with buffers of size 1
- Run A, B, C, D



22

Parks' Algorithm in Action

- B blocked waiting for space in B->C buffer
- Run A, then C
- System will run indefinitely



23

Parks' Scheduling Algorithm

- Neat trick
- Whether a Kahn network can execute in bounded memory is undecidable
- Parks' algorithm does not violate this
- It will run in bounded memory if possible, and use unbounded memory if necessary

24

Using Parks' Scheduling Algorithm

- It works, but…

- Requires dynamic memory allocation
- Does not guarantee minimum memory usage
- Scheduling choices may affect memory usage
- Data-dependent decisions may affect memory usage
- Relatively costly scheduling technique
- Detecting deadlock may be difficult

25

Kahn Process Networks

- Their beauty is that the scheduling algorithm does not affect their functional behavior
- Difficult to schedule because of need to balance relative process rates
- System inherently gives the scheduler few hints about appropriate rates
- Parks' algorithm expensive and fussy to implement
- Might be appropriate for coarse-grain systems
  - Scheduling overhead dwarfed by process behavior

26

Synchronous Dataflow (SDF)

- Edward Lee and David Messerchmitt, Berkeley, 1987
- Restriction of Kahn Networks to allow compile-time scheduling
- Basic idea: each process reads and writes a fixed number of tokens each time it fires:

loop

read 3 A, 5 B, 1 C … compute … write 2 D, 1 E, 7 F

end loop

27

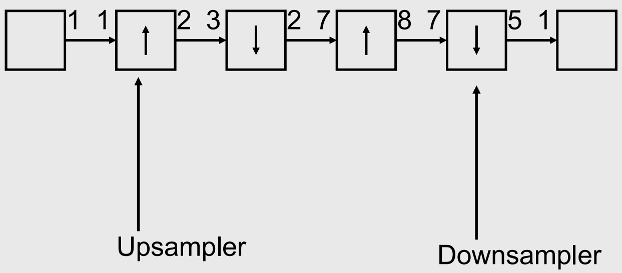
SDF and Signal Processing

- Restriction natural for multirate signal processing
- Typical signal-processing processes:
- Unit-rate
  - Adders, multipliers
- Upsamplers (1 in, n out)
- Downsamplers (n in, 1 out)

28

Multi-rate SDF System

- DAT-to-CD rate converter
- Converts a 44.1 kHz sampling rate to 48 kHz



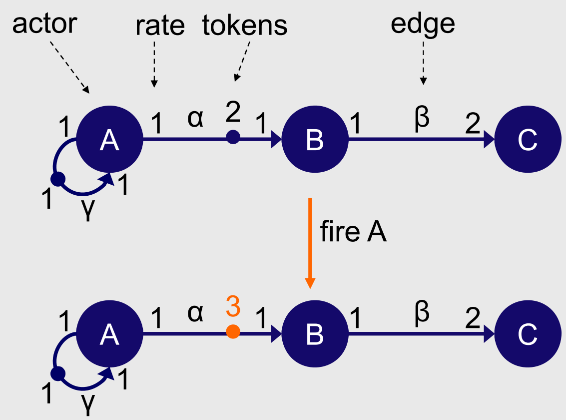
29

Delays

- Kahn processes often have an initialization phase
- SDF doesn't allow this because rates are not always constant
- Alternative: an SDF system may start with tokens in its buffers
- These behave like delays (signal-processing)
- Delays are sometimes necessary to avoid deadlock

30

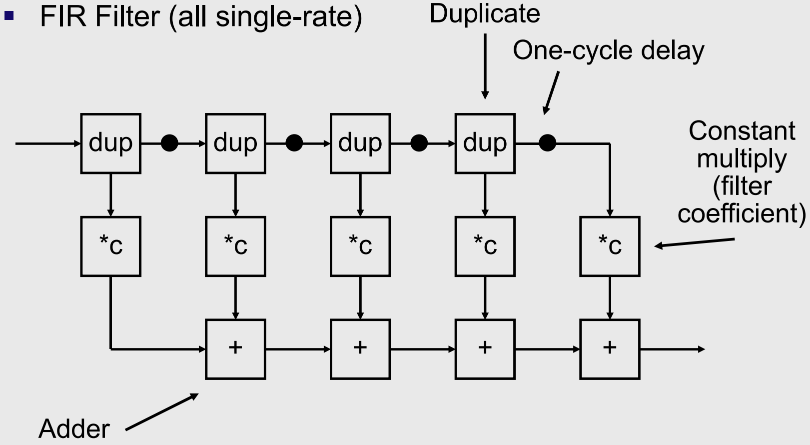
Synchronous Dataflow Graphs (SDFGs)



31

Example SDF System

- FIR Filter (all single-rate)



32

SDF Scheduling

- Schedule can be determined completely before the system runs

- Two steps:

1. Establish relative execution rates by solving a system of linear equations
2. Determine periodic schedule by simulating system for a single round

33

SDF Scheduling

- Goal: a sequence of process firings that

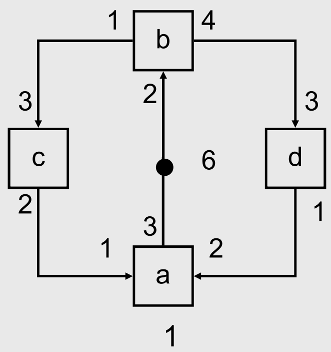
- Runs each process at least once in proportion to its rate
- Avoids underflow
  - no process fired unless all tokens it consumes are available
- Returns the number of tokens in each buffer to their initial state

- Result: the schedule can be executed repeatedly without accumulating tokens in buffers

34

Calculating Rates

- Each arc imposes a constraint



$3a - 2b = 0$

$4b - 3d = 0$

$b - 3c = 0$

$2c - a = 0$

$d - 2a = 0$

Solution:

$a = 2c$

$b = 3c$

$d = 4c$

35

Calculating Rates

- Consistent systems have a one-dimensional solution
  - Usually want the smallest integer solution
    → ***Repetition vector***

- Inconsistent systems only have the all-zeros solution

- Disconnected systems have two- or higher-dimensional solutions

36

Calculating Repetition Vector

- MCM Algorithm (poly complexity)

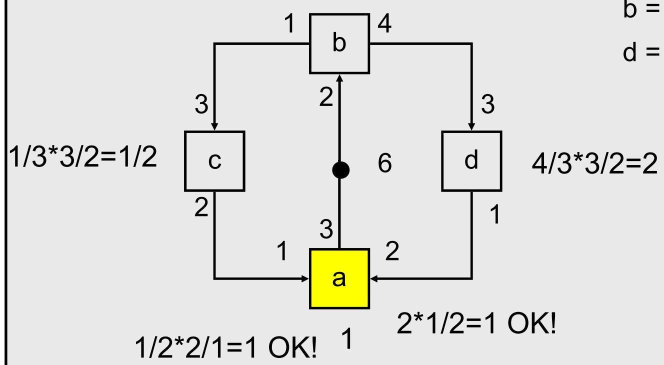
Balance equations:

a = 2c

b = 3c

d = 4c

Rb=Ra*Rh/Rt 3/2

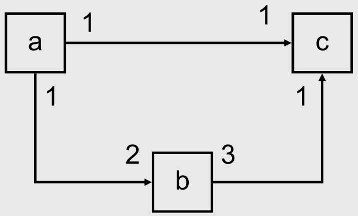


mcm=2 → Iteration vector [A:2, B:3, C:1, D:4]

37

An Inconsistent System

- No way to execute it without an unbounded accumulation of tokens
- Only consistent solution is "do nothing"



$a - c = 0$

$a - 2b = 0$

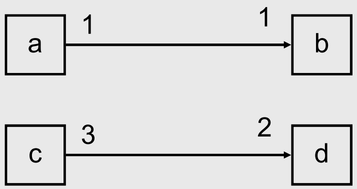
$3b - c = 0$

$3a - 2c = 0$

38

An Underconstrained System

- Two or more unconnected pieces
- Relative rates between pieces undefined



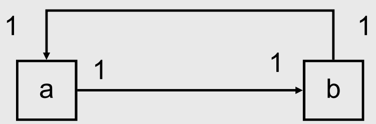
$a - b = 0$

$3c - 2d = 0$

39

Consistent Rates Not Enough

- A consistent system with no schedule
- Rates do not avoid deadlock



- Solution here: add a delay on one of the arcs

40

SDF Scheduling

- Fundamental SDF Scheduling Theorem:

If rates can be established, any scheduling algorithm that avoids buffer underflow will produce a correct schedule if it exists (Periodic Admissible Seq Schedule)

ALGO PASS

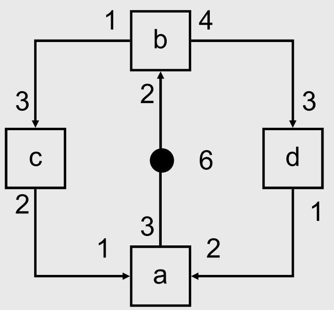
1. Compute repetition vector q
2. Form an arbitrarily ordered list L of all nodes
3. For each n in L, schedule n if it is runnable, trying each n once
4. If each n has been scheduled qn times, STOP
5. If no node can be scheduled DEADLOCK
6. Go to 3

Use q → MINIMUM # of task executions!

41

Scheduling Example

- Theorem guarantees any valid simulation will produce a schedule



a=2 b=3 c=1 d=4

Possible schedules:

BBBCDDDDAA

BDBDBCADDA

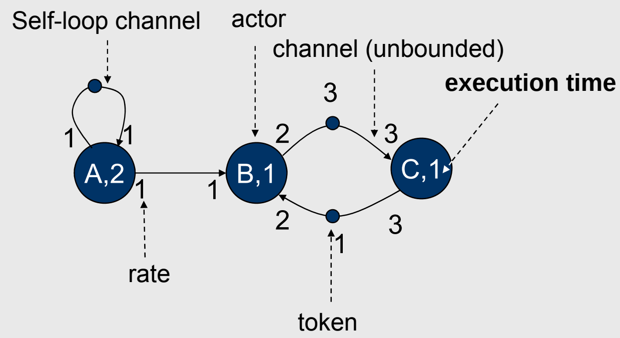
BBDDBDDCAA

… many more

BC … is not valid

42

Timed SDFG



Single processor schedule using q → MINIMUM LATENCY!

43 **Throughput Definition**

- Actor throughput:

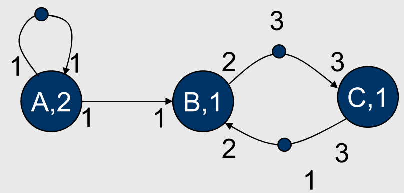
  The ***average number of firings*** of one actor ***per time unit***

$$Th(a) = \lim_{k\to\infty} \frac{k \text{ firings of } a}{\text{end time of these firings}}.$$

- (Normalized) graph throughput (if SDFG is consistent):

$$\min_{\text{actors } a} \frac{Th(a)}{q(a)}.$$

44 **Computing throughput for PASS**



q=[(A,3), (B, 3), (C, 2)] $\xrightarrow{\text{PASS}}$ ACABABCB

Th(A)=3/(2+1+2+1+2+1+1+1)=3/(3*2+1*3+1*3)=3/12

Th(B)=3/12, Th(C)=2/12

Th(SDG)=1/12

Single processor schedule using q → MINIMUM LATENCY!

45

Scheduling Choices

- SDF Scheduling Theorem guarantees a schedule will be found if it exists
- Systems often have many possible schedules
- How can we use this flexibility?
  - Reduced code size
  - Reduced buffer sizes

46

SDF Code Generation (single core scheduling)

- Often done with prewritten blocks
- For traditional DSP, handwritten implementation of large functions (e.g., FFT)
- One copy of each block's code made for each appearance in the schedule
  - I.e., no function calls

47

Code Generation

- In this simple-minded approach, the schedule

BBBCDDDDAA

would produce code like

```
B;
B;
C;
D;
D;
D;
D;
A;
A;
```

48

Looped Code Generation

- Obvious improvement: use loops

- Rewrite the schedule in "looped" form:

(3 B) C (4 D) (2 A)

- Generated code becomes

```
for ( i = 0 ; i < 3; i++) B;
C;
for ( i = 0 ; i < 4 ; i++) D;
for ( i = 0 ; i < 2 ; i++) A;
```

49

Single-Appearance Schedules

- Often possible to choose a looped schedule in which each block appears exactly once
- Leads to efficient block-structured code
  - Only requires one copy of each block’s code
- Does not always exist
- Often requires more buffer space than other schedules

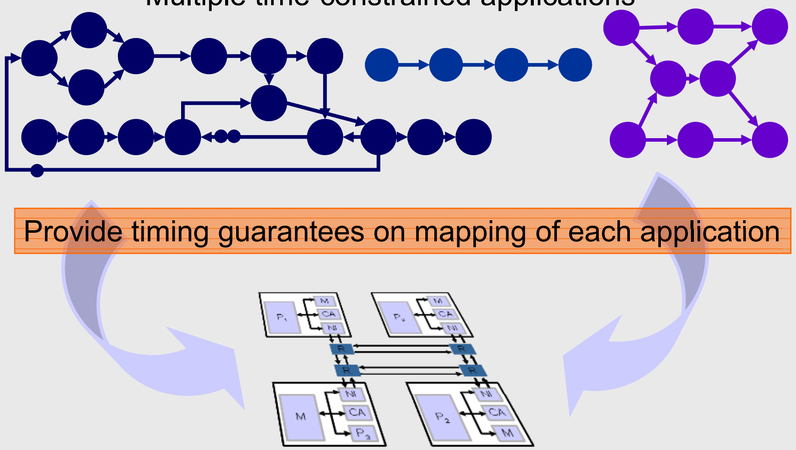
50

Minimum-Memory Schedules

- Another possible objective
- Often increases code size (block-generated code)
- Static scheduling makes it possible to exactly predict memory requirements

51 **Mapping onto MPSoC Platforms**

Multiple time-constrained applications



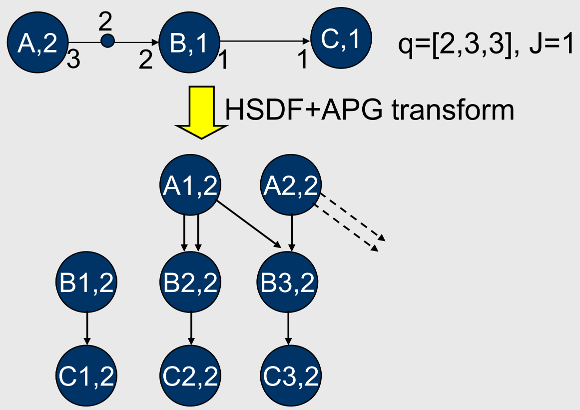
Provide timing guarantees on mapping of each application

Multiprocessor system

52 **Parallel (multi-core) schedules**

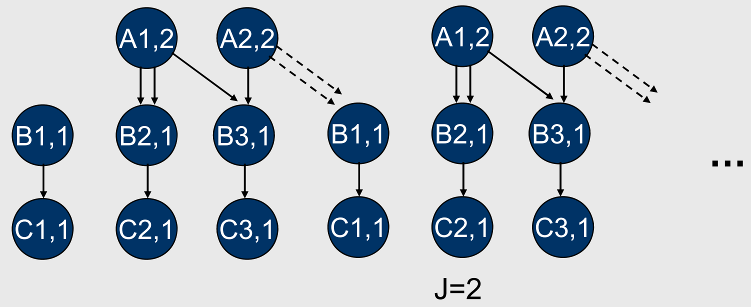
- Given Ps (smallest possible PASS period), J (unroll multiplicative factor on period)
- Convert SDF into HSDF, then into an Acyclic Precedence Graph (APG) while unrolling it J times
  - The three steps can be performed in sequence
- Schedule the APG for minimum makespan (assuming that max througput is the target), taking into account resource constraints

53 **Example**



- Big catch… exponential blowup in #nodes is possible!

54 **Unrolling…**



- J=1, Th=1/4
- J=2, Th=2/4
- J=n, Th=n/4

55

Is speedup unbounded?

- NO! Every SDF has a maximum speedup, called *MCM bound*
- The bound can be efficiently computed on HSDF
- The minimum iteration period T:

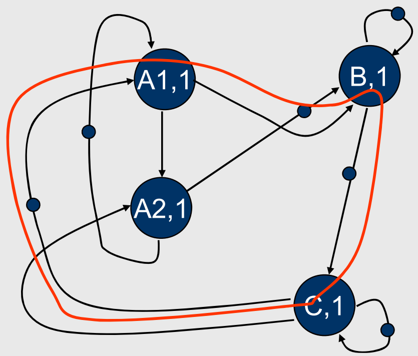
$$T = \max_{cycle \in SDF} \left\{ \frac{\sum_{v \in cycle} t(v)}{D(cycle)} \right\}$$

- This is given *unbounded resources*
  - NOTE: if there are no loops T→0

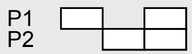
56

Example

- HSDF (from SDF)



- T= (1+1+1)/1=3



Periodic schedule that achieves MCM bound

- Polynomial-time computation on HSDF

57

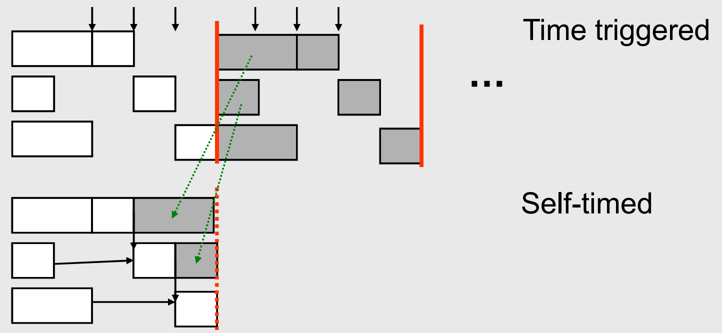
Achieving the MCM bound

- Can be achieved with a periodic time-triggered schedule (everything is synchronized) by optimal unroling J_{OPT}
  - J_{OPT} can be determined by a transformation [Parhi91]
    - SDF → HSDF
    - Unfold HSDF mcm(delays in loops) times
  - May Imply a big increase in task execution instances (node blowup)
- Can be achieved with a *self-timed schedule*
  - Execute each node ASAP when it is enabled!
  - It can be demonstrated that a self-timed schedule has the following structure:
    - Finite sequence of firings – non periodic part
    - Infinite sequence of firiring – periodic part
  - Implementation of STS can be tricky (…but)

58

Time-triggered vs. Self-timed schedule

- Different execution model: timers vs. synchronization



- Iterations are naturally partially overlapped
- It handles un-certain execution times
- Works also with limited resources $T_{ST} \leq T_{TT}$

Motivation for Direct-SDFG techniques

- Existing techniques use homogeneous SDFGs
- Throughput analysis may be very slow for realistic applications when using homogeneous SDFGs
  - Potential exponential blowup!
- Use SDFGs for resource allocation and throughput analysis

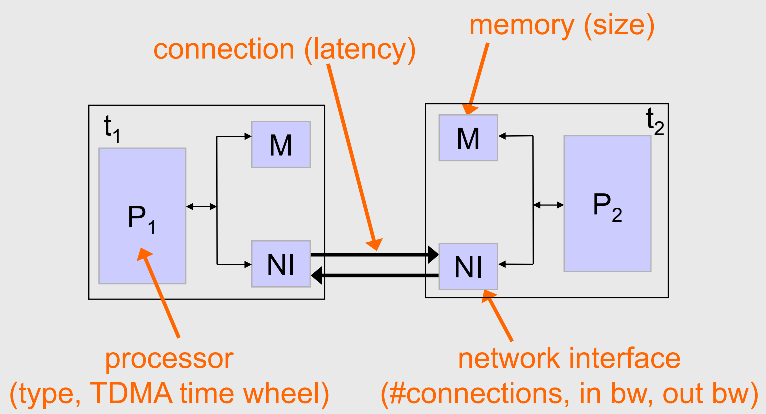
60

Scheduling

- Processors shared between actors or applications
  - Timing guarantee for each application individually
  - Minimize resource usage for each application
- TDMA scheduling
  - Independent timing behavior between tasks
  - Potentially large resource reservations
- Static-order scheduling
  - Over-allocation of resources is limited
  - Ordering of tasks must be known a-priori
- TDMA scheduling between applications
- Static-order scheduling between actors of an application

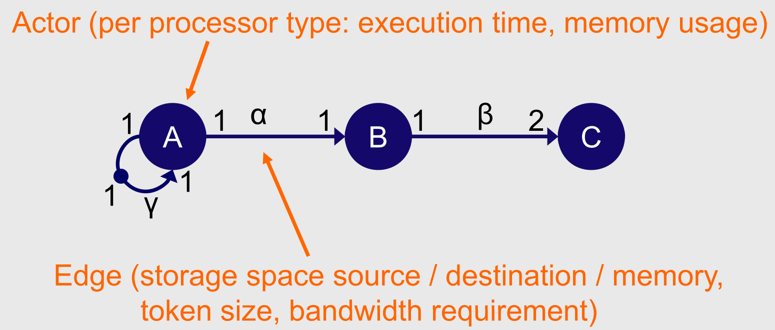
Architecture platform

- Heterogeneous tile-based architecture

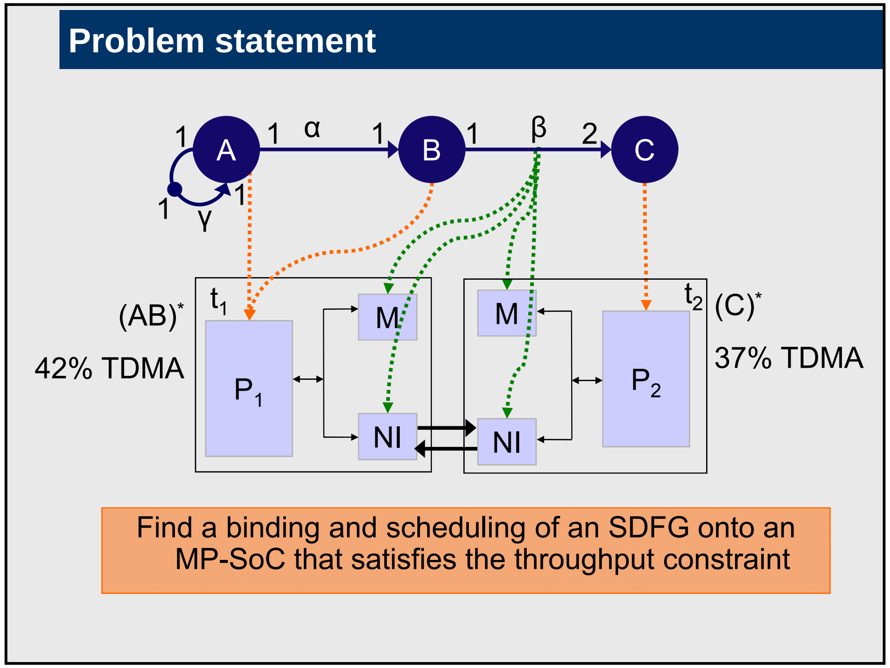


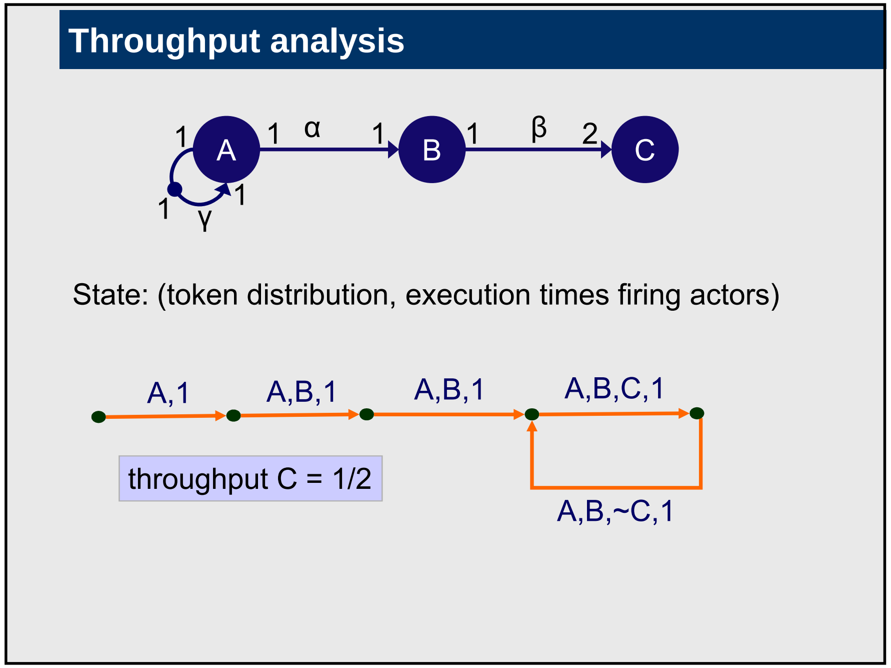
Streaming application graph

- Application modeled with SDFG

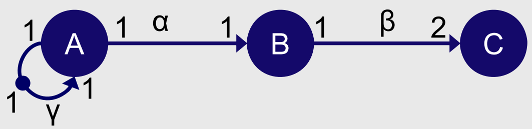


Throughput constraint on graph

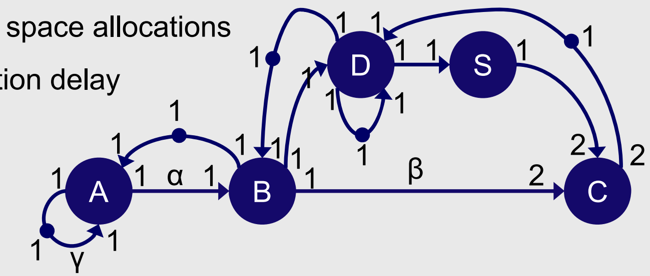

Problem statement
1
A
1
α
1
B
1
β
2
C
1
γ
1
(AB)*
42% TDMA
t1
P1
M
NI
M
NI
P2
t2
(C)*
37% TDMA
Find a binding and scheduling of an SDFG onto an MP-SoC that satisfies the throughput constraint


Throughput analysis
1
A
1
α
1
B
1
β
2
C
1
γ
1
State: (token distribution, execution times firing actors)
A,1
A,B,1
A,B,1
A,B,C,1
throughput C = 1/2
A,B,~C,1

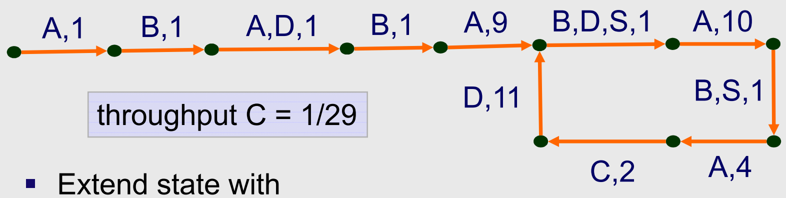
Binding-aware SDFG



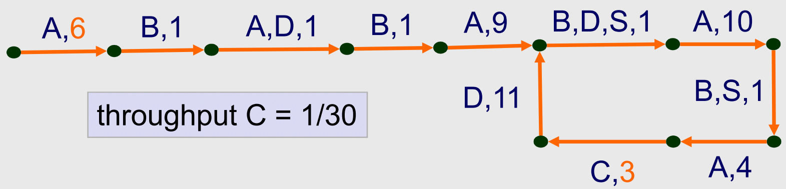
- Model in SDFG
  - TDMA time wheel synchronization
  - storage space allocations
  - connection delay



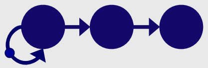
Throughput analysis



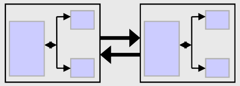
- Extend state with
  - position of static-order schedule
  - position TDMA time wheel



Resource allocation strategy



Throughput-constrained SDFG



MP-SoC architecture

Actor binding

↓

Static-order scheduling

↓

Time slice allocation

↓

Bound and scheduled SDFG

Actor binding

- Actors sorted on “criticality”
  - Related to notion of Cycle-Mean in HSDF
- Binding considers
  - Processing load
  - Memory load
  - Communication load
- Cost function weights alternatives

$$\text{cost}(t) = c_1 \cdot l_p(t) + c_2 \cdot l_m(t) + c_3 \cdot l_c(t)$$

Static-order scheduling

- Order actor firings of an application on a processor
- List-scheduling algorithm

Time slice allocation

- Provide timing independence between applications
- Binary search algorithm using fast throughput analysis technique

Experimental setup

- Architecture
  - 3x3 mesh of tiles
  - 3 different processor types
- Four sets of three sequences of SDFGs
  - Compute intensive
  - Memory intensive
  - Communication intensive
  - Balanced
- Sequence of SDFGs bound to architecture till no valid binding can be found for an SDFG

Experimental results

cost	compute intensive	memory intensive	communication intensive	balanced
1,0,0	20.22	5.22	7.56	18.56
0,1,0	18.78	8.00	11.33	23.33
0,0,1	29.22	7.56	12.89	25.00
1,1,1	18.44	6.50	10.33	23.56
0,1,2	24.56	8.00	12.89	30.11

Ip, Im, Ic

- 16.1 throughput computations per SDFG

Experimental results

- Application
  - 3x H.263 decoders (4 actors)
  - 1x MP3 decoder (13 actors)
- Architecture
  - 2x2 mesh of tiles
  - 2 accelerators, 2 general-purpose processors
- Cost function (2,0,1)
  - Focus on processing and communication
- 34 throughput computations
- Run-time 8 minutes

Conclusions

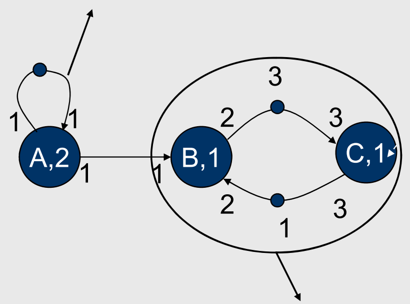
- Resource allocation strategy for SDFGs on MP-SoCs
- Most expressive model-of-computation used so far
- Technique provides timing guarantees
- Cost functions can steer resource allocation
- Experiments show feasibility of the approach

75

Understanding the MCM bound

Cycle 1: 3*2/1

q=[(A,3), (B, 3), (C, 2)]



Cycle 2: (1*3+1*2)/(3/3+1/2)

- Using a generalized formula for the computation of MCM (equivalent to the