Lecture 5
Dataflow Process Models

Stephen A. Edwards
Forrest Brewer
Ryan Kastner
Philosophy of Dataflow Languages

- 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
Dataflow Languages

- Every process can run concurrently
  - Processes side-effect free resources assumed
- Processes described with imperative code
  - FSM, N DFA model of hardware or software
- Processes *only* communicate through buffers
  - Both control and data
- Parallelism is bounded by places and data-flow
  - Can describe general purpose computation this way
    - Requires alternative viewpoint and metrics
  - Fits transactional models of a system
    - Data-base (Google)
- Execution driven by demand
Dataflow Language Model

- Processes communicating through FIFO buffers
Dataflow Communication

- Communication *only* through buffers
  - No side effects (or shared memory)
- Buffers are unbounded for simplicity
  - Causes model complexity issues
- Token Sequence into link is sequence out of link
  - Links are strictly FIFO
- Destructive read: reading a value from a buffer removes the value
  - Cannot ‘check’ to see new token without read
- Unlike shared memory, can always determine latency
Applications of Dataflow Models

- Poor fit for a word processor
  - Data-flow models are weak on control intensive behavior
- Common in signal-processing applications
  - Ordered streams of data
  - Simple map to pipelined hardware
    - Lab View, Simulink, System C Transactions
- Buffers used for signal processing applications anyway
  - FIFO buffers allow for mediation of bursty flows up to capacity of the buffer
    - Rates must strictly agree on average
Applications of Dataflow

- Good fit for block-diagram specifications
  - System Level RTL (directed links)
  - Linear/nonlinear control systems (Feedback Networks)
  - Network Computing
- Common in Electrical Engineering
- Value: reasoning about data rates, availability, latency and performance can be done abstractly
  - Used for top-level models before processes are designed
  - Allow reasoning about process requirements
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
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
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
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
A Kahn Process

- From Kahn’s original 1974 paper

```c
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
A Kahn Process

- From Kahn’s original 1974 paper

```plaintext
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
A Kahn Process

- From Kahn’s original 1974 paper

```c
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
Possible Runs of Kahn System

- Starts from upper left corner
- Deterministic since all output writes must cross boundary
  - Left going arcs ‘0’
  - Right going arcs ‘1’

Thus all possible output sequences alternate 0/1/0…
Determinacy

- Process: “ordered mapping” of input sequence to output sequences
- Continuity: process uses prefix of input sequences to produce prefix of output sequences. Adding more tokens does not change the tokens already produced
- The state of each process depends on token values rather than their arrival time
- Unbounded FIFO: the speed of the two processes does not affect the sequence of data values
  - Practical networks need to mind this well
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
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
Routes to Nondeterminism

- Allow processes to test for emptiness
  - If the token behavior changes, violates monotonic property
  - Cannot choose from possible inputs (i.e. if token on either input… is not legal)
- Allow processes themselves to be nondeterminate
- Allow more than one process to read from a channel
  - Cannot solve precedence issues in general
- Allow more than one process to write to a channel
  - Cannot fix the order of processes on channel
- Allow processes to share a variable
  - Unbounded communication bandwidth can cause several problems above…
Scheduling Kahn Networks

- Challenge is running processes without accumulating tokens

Diagram:

- Node A
- Node B
- Node C
- A -> C
- B -> C
Scheduling Kahn Networks

- Challenge is running processes without accumulating tokens

Diagram:

- Node A
  - Sends tokens to Node C
  - Only consumes tokens from Node A

- Node B
  - Receives tokens from Node A
  - Tokens will accumulate here
  - Always emit tokens

- Node C
  - Receives tokens from Node A
  - Only consumes tokens from Node A
Demand-driven Scheduling?

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

```
A --> C
B --> D
```

Always produce tokens
Always consume tokens
Other Difficult Systems

Not all systems can be scheduled without token accumulation

Produce two a's for every b
Alternates between receiving one a and one b
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
Parks’ Algorithm in Action

- Start with buffers of size 1
- Run A, B, C, D
Parks’ Algorithm in Action

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

- Tokens $\rightarrow$ Data
- Assignment $\rightarrow$ Placing a token in the output arc
- Snapshot / configuration: state
- Computation
  - The intermediate step between snapshots / configurations
- An actor of a dataflow graph is enabled if there is a token on each of its input arcs
Synchronous Dataflow (SDF)
Fixed Production/Consumption Rates

- Balance equations (one for each channel):
  \[ f_A N = f_B M \]

- Schedulable statically
- Get a well-defined “iteration”
- Decidable:
  - buffer memory requirements
  - deadlock

```
fire A {
    ...
    produce N
    ...
}
```

```
fire B {
    ...
    consume M
    ...
}
```
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)
Operational Semantics
Firing Rule

- Any enabled actor may be fired to define the “next state” of the computation
- An actor is fired by removing a token from each of its input arcs and placing tokens on each of its output arcs.
- Computation ➔ A Sequence of Snapshots
  - Many possible sequences as long as firing rules are obeyed
  - Determinacy
  - “Locality of effect”
Multi-rate SDF System

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

- FIR Filter (all single-rate)

Diagram:
- Duplication
- One-cycle delay
- Constant multiply (filter coefficient)
- Adder
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
Balance equations

• Number of produced tokens must equal number of consumed tokens on every edge

\[ \text{A} \quad n_p \quad \rightarrow \quad n_c \quad \text{B} \]

• Repetitions (or firing) vector \( v_S \) of schedule \( S \): number of firings of each actor in \( S \)

• \( v_S(A) \) \( n_p = v_S(B) \) \( n_c \)

must be satisfied for each edge
Balance equations

- Balance for each edge:
  - $3 v_S(A) - v_S(B) = 0$
  - $v_S(B) - v_S(C) = 0$
  - $2 v_S(A) - v_S(C) = 0$
  - $2 v_S(A) - v_S(C) = 0$
Balance equations

\[
M v_S = 0
\]

iff S is periodic

• Full rank (as in this case)
  • no non-zero solution
  • no periodic schedule

(too many tokens accumulate on A->B or B->C)
Balance equations

- Non-full rank
  - infinite solutions exist (linear space of dimension 1)
- Any multiple of \( q = |1 \ 2 \ 2|^{\top} \) satisfies the balance equations
- ABCBC and ABBCC are minimal valid schedules
- ABABBCBCCC is non-minimal valid schedule

\[
M = \begin{bmatrix} 2 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \end{bmatrix}
\]
Static SDF scheduling

- Main SDF scheduling theorem (Lee ‘86):
  - A connected SDF graph with $n$ actors has a periodic schedule iff its topology matrix $M$ has rank $n-1$
  - If $M$ has rank $n-1$ then there exists a unique smallest integer solution $q$ to
    \[ Mq = 0 \]

- Rank must be at least $n-1$ because we need at least $n-1$ edges (connected-ness), providing each a linearly independent row

- Admissibility is not guaranteed, and depends on initial tokens on cycles
Admissibility of schedules

- No admissible schedule: BACBA, then deadlock...
- Adding one token on A->C makes BACBACBA valid
- Making a periodic schedule admissible is always possible, but changes specification...
From repetition vector to schedule

- Repeatedly schedule fireable actors up to number of times in repetition vector
  \[ q = [1 \quad 2 \quad 2]^T \]

- Can find either ABCBC or ABBCC

- If deadlock before original state, no valid schedule exists (Lee ‘86)
Calculating Rates

- Each arc imposes a constraint

\[
\begin{align*}
3a - 2b &= 0 \\
4b - 3d &= 0 \\
b - 3c &= 0 \\
2c - a &= 0 \\
d - 2a &= 0
\end{align*}
\]

Solution?

\[
\begin{align*}
a &= 2c \\
b &= 3c \\
d &= 4c
\end{align*}
\]
Calculating Rates

- Consistent systems have a one-dimensional solution
  - Usually want the smallest integer solution

- Inconsistent systems only have the all-zeros solution

- Disconnected systems have two- or higher-dimensional solutions
An Inconsistent System

- No way to execute it without an unbounded accumulation of tokens
- Only consistent solution is “do nothing”
An Underconstrained System

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

\[ a - b = 0 \]
\[ 3c - 2d = 0 \]
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
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
Scheduling Example

- Theorem guarantees any valid simulation will produce a schedule

\[ a=2 \quad b=3 \quad c=1 \quad d=4 \]

Possible schedules:
- BBBDDDDAA
- BDBDBCADDA
- BBDBBDDCAAA
- \ldots \text{many more}

BC \ldots \text{is not valid}
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
Schedules

- Dash is single appearance schedule
- Short Dash is minimum buffer schedule
- Note: SDF schedules form a lattice
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
SDF Code Generation

- 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
Code Generation

- In this simple-minded approach, the schedule
  
  BBBCCDDDDAA
  
would produce code like

  B;
  B;
  B;
  C;
  D;
  D;
  D;
  D;
  A;
  A;
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;
  ```
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
Finding Single-Appearance Schedules

- Always exist for acyclic graphs
  - Blocks appear in topological order
- For SCCs, look at number of tokens that pass through arc in each period (follows from balance equations)
- If there is at least that much delay, the arc does not impose ordering constraints
- Idea: no possibility of underflow

\[ a = 2 \quad b = 3 \]

6 tokens cross the arc
delay of 6 is enough
Finding Single-Appearance Schedules

- Recursive strongly-connected component decomposition
- Decompose into SCCs
- Remove non-constraining arcs
- Recurse if possible
  - Removing arcs may break the SCC into two or more
Minimum-Memory Schedules

- Another possible objective

- Often increases code size (block-generated code)

- Static scheduling makes it possible to exactly predict memory requirements

- Simultaneously improving code size, memory requirements, sharing buffers, etc. remain open research problems
Cyclo-static Dataflow

- SDF suffers from requiring each process to produce and consume all tokens in a single firing
- Tends to lead to larger buffer requirements
- Example: downsampler

- Don’t really need to store 8 tokens in the buffer
- This process simply discards 7 of them, anyway
Cyclo-static Dataflow

- Alternative: have periodic, binary firings

\[
\{1,1,1,1,1,1,1,1\} \xrightarrow{\text{down}} \{1,0,0,0,0,0,0,0\}
\]

- Semantics: first firing: consume 1, produce 1
- Second through eighth firing: consume 1, produce 0
Cyclo-Static Dataflow

- Scheduling is much like SDF
- Balance equations establish relative rates as before
- Any scheduler that avoids underflow will produce a schedule if one exists
- Advantage: even more schedule flexibility
- Makes it easier to avoid large buffers
- Especially good for hardware implementation:
  - Hardware likes moving single values at a time
Cyclostatic Dataflow (CSDF)
(Lauwereins et al., TU Leuven, 1994)

- Actors cycle through a regular production/consumption pattern.
- Balance equations become:

\[ f_A \sum_{i=0}^{R-1} N_{i \mod P} = f_B \sum_{i=0}^{R-1} M_{i \mod Q}; \quad R = \text{lcm}(P, Q) \]
Cyclo-Static Dataflow

- Scheduling similar to SDF
- Balance equations establish relative rates
- Key: avoid underflow of channel
- Advantages
  - Increased schedule flexibility
    - Easier to avoid large buffers
  - Closer to parallel hardware model
    - Links move single values at a time
Multidimensional SDF
(Lee, 1993)

- Production and consumption of $N$-dimensional arrays of data:

- Balance equations and scheduling policies generalize.
- Much more data parallelism is exposed.
Boolean and Integer Dataflow (BDF, IDF) (Lee and Buck, 1993)

- Balance equations are solved symbolically in terms of unknowns that become known at run time.
- An annotated schedule is constructed with predicates guarding each action.
- Existence of such an annotated schedule is undecidable (as is deadlock & bounded memory)
  - However often can check efficiently

\[
\begin{align*}
    f_{\text{switch}}b &= f_B \\
    f_{\text{switch}}(1 - b) &= f_C
\end{align*}
\]
Undecidability
(Buck ’93)

• Sufficient set of actors for undecidability:
  - boolean functions on boolean tokens
  - switch and select
  - initial tokens on arcs

• Undecidable:
  - deadlock
  - bounded buffer memory
  - existence of an annotated schedule
Dynamic Dataflow (DDF)

- Actors have firing rules
  - Data consumed/produced may vary depending on the values
  - Set of finite prefixes on input sequences
  - Firing function applied to finite prefixes yield finite outputs

- Scheduling objectives:
  - Do not stop if there are executable actors
  - Execute in bounded memory if this is possible
  - Maintain determinacy if possible

- Policies that fail:
  - Data-driven execution
  - Demand-driven execution
  - Fair execution
  - Many balanced data/demand-driven strategies

- Policy that succeeds (Parks 1995):
  - Execute with bounded buffers
  - Increase bounds only when deadlock occurs
Summary of Dataflow

- Processes communicating exclusively through FIFOs

- Kahn process networks
  - Blocking read, nonblocking write
  - Deterministic
  - Hard to schedule
  - Parks’ algorithm requires deadlock detection, dynamic buffer-size adjustment
Summary of Dataflow

- Synchronous Dataflow (SDF)
- Firing rules:
  - Fixed token consumption/production
- Can be scheduled statically
  - Solve balance equations to establish rates
  - Any correct simulation will produce a schedule if one exists
- Looped schedules
  - For code generation: implies loops in generated code
  - Recursive SCC Decomposition
- CSDF: breaks firing rules into smaller pieces
  - Scheduling problem largely the same