Dataflow Languages

Prof. Stephen A. Edwards

Dataflow Language Model

- Processes communicating through FIFO buffers

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

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 Language Model

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

Dataflow Communication

- Once proposed for general-purpose programming
- Fundamentally concurrent: should map more easily to parallel hardware
- A few lunatics built general-purpose dataflow computers based on this idea
- Largely a failure: memory spaces anathema to the dataflow formalism
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

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 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 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("%d", i);
    send(i, w);
    b = !b;
  }
}
```

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

Applications of Dataflow

- Perfect fit for block-diagram specifications
  - Circuit diagrams
  - Linear/nonlinear control systems
  - Signal processing

- Suggest dataflow semantics

- Common in Electrical Engineering
- Processes are blocks, connections are buffers
A Kahn Process

- From Kahn’s original 1974 paper

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

Process interface includes FIFOs

Send() writes a data value on an output FIFO

A Kahn System

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

```
A Kahn System

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

Emits a 1 then copies input to output

\[ h \rightarrow g \rightarrow f \rightarrow h \]

Emits a 0 then copies input to output
```

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

Scheduling Kahn Networks

- Challenge is running processes without accumulating tokens

```
Scheduling Kahn Networks

- Challenge is running processes without accumulating tokens

\[ A \rightarrow B \rightarrow C \]
```
Scheduling Kahn Networks

- Challenge is running processes without accumulating tokens

Demand-driven Scheduling?

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

Other Difficult Systems

- Not all systems can be scheduled without token accumulation

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

**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)

**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)

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

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:
\[
\begin{align*}
a &= 2c \\
b &= 3c \\
d &= 4c
\end{align*}
\]
An Inconsistent System

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

\[ a - c = 0 \]
\[ a - 2b = 0 \]
\[ 3b - c = 0 \]
\[ 3a - 2c = 0 \]

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:

BBBCDDDA
BDDBDCA
BBDDBDCAA
… many more

BC … is not valid

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;
  C;
  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

\[
\begin{array}{c}
\text{b} \\
\text{2} \\
\text{3} \\
\text{a} \\
\end{array}
\]

6 tokens cross the arc
delay of 6 is enough
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

```
8
\downarrow
1
```

- 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\} \rightarrow \{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

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