Dataflow Languages

Languages for Embedded Systems
Prof. Stephen A. Edwards
Summer 2005
NCTU, Taiwan

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 Language Model

Processes communicating through FIFO buffers

Dataflow Languages

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

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

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

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 Process from Kahn’s 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(v);
        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

---

**Another Sample Process**

```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

---

**A Kahn System**

Prints an alternating sequence of 0s and 1s

---

**Proof of Determinism**

Because a process cannot 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:

Imagine you are a process. You are only affected by the sequence of tokens on my inputs, and can’t tell whether they arrive early, late, or in what order (blocking takes care of this, but you can’t tell whether you blocked).

You will behave the same in any case

Thus, the sequence of tokens you put on your outputs is the same regardless of the timing of the tokens on your inputs

---

**Scheduling Kahn Networks**

Challenge is running processes without accumulating tokens
Scheduling Kahn Networks

Challenge is running processes without accumulating tokens

Only consumes tokens from A

Tokens will accumulate here

Always emit tokens

Demand-driven Scheduling?

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

Always emits tokens

Always consumes tokens

Other Difficult Systems

Not all systems can be scheduled without token accumulation

Produces 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

Run A

Run B

Run C

Run D

Run A

Run C

B blocked waiting for space in B → C buffer

Run A, then C, then 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 algorithm
- 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 where 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:

\[
\text{loop} \\
\text{read 3 A, 5 B, 1 C} \ldots \text{compute} \ldots \text{write 2 D, 1 E, 7 F} \\
\text{end loop}
\]

SDF and Signal Processing

Restriction natural for multirate signal processing

Typical signal-processing processes:
- Unit-rate
  - e.g., Adders, multipliers
- Upsamplers (1 in, n out)
- Downsamplers (n in, 1 out)

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 signal-processing-like delays

Delays are sometimes necessary to avoid deadlock

Example SDF System

FIR Filter (all unit rate)

\[
\begin{align*}
\text{dup} & \rightarrow \text{dup} \\
\times c & \rightarrow \times c \\
+ & \rightarrow + \\
\end{align*}
\]

Delays are sometimes necessary to avoid deadlock

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

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:

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

Multi-rate SDF System

DAT-to-CD rate converter

Converts a 44.1 kHz sampling rate to 48 kHz

\[
\begin{align*}
1 &\rightarrow 1 \\
2 &\rightarrow 3 \\
7 &\rightarrow 8 \\
5 &\rightarrow 1 \\
\end{align*}
\]

Upsampler

Downsampler

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 to do nothing

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

Implies

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

**An Underconstrained System**

Two or more unconnected pieces

Relative rates between pieces undefined

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

**Consistent Rates Are Not Enough**

A consistent system with no schedule

Rates do not prevent 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, provided one exists

**Scheduling Example**

```
1 4 3 2 1
```

Possible schedules:

BBBCDDDDAA

BBDDBCADDA

BBDDBDCCAA

... 

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?

To reduce code size

To reduce buffer sizes

**SDF Code Generation**

Often done with prewritten blocks inlined according to the schedule

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

```
BBBCDDDDAA
```

would produce code like

```
B; B; B;
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;
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 SASs

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

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
Semantics: first firing: consume 1, produce 1
Second through eighth firing: consume 1, produce 0

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
• A correct simulation produces 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. Similar scheduling technique