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
Dataflow Model of Computation

- Processes communicating through FIFO buffers

Dataflow Semantics

- 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
### 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

### 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 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(v);
        printf("%i\n", i);
        send(i, w);
        b = !b;
    }
}
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 reads from u and alternately copies it to v and w

---

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
A Kahn System

- Prints an alternating sequence of 0's and 1's
  - Emits a 1 then copies input to output
    - h
  - Emits a 0 then copies input to output
    - g
    - f
    - h

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

![Diagram of Kahn Network]

- A
- B
- C
Scheduling Kahn Networks

- Challenge is running processes without accumulating tokens

![Diagram showing Kahn Network with nodes A, B, C and arrows indicating flow and token consumption/production]

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

![Diagram of a process network with two processes connected by a directed edge labeled 'a' and 'b'.]

- 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

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

```
R ABCD
```

Only consumes tokens from A

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

```
A
```

```
C
```

```
B
```

```
D
```

```
A
```

```
C
```

```
B
```

```
D
```
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

![Diagram of multi-rate SDF System]
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

Synchronous Dataflow Graphs (SDFGs)

[Diagram showing data flow and token movement through actors and edges]
Example SDF System

- FIR Filter (all single-rate)
  
  Duplicate
  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
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:
- \(a = 2c\)
- \(b = 3c\)
- \(d = 4c\)
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

Calculating Repetition Vector

- MCM Algorithm (poly complexity)

Balance equations:

\[ R_b = R_a * R_h / R_t \]

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

\[ \frac{3}{2} \]

\[ \frac{1}{3} * \frac{3}{2} = \frac{1}{2} \]
\[ \frac{2}{3} * \frac{1}{2} = \frac{1}{2} \text{ OK!} \]
\[ \frac{1}{2} * \frac{2}{1} = 1 \text{ OK!} \]
\[ \frac{2}{1} * \frac{1}{2} = 1 \text{ OK!} \]

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

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

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

An Underconstrained System

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

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

\[
\begin{align*}
a &- 1 \\
b &- 1 \\
c &- 3 \\
d &- 2
\end{align*}
\]
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 (Periodic Admissible Seq Schedule)

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 \( q \to \) MINIMUM # of task executions!
Scheduling Example

- Theorem guarantees any valid simulation will produce a schedule

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

Possible schedules:
- BBBCDDDDAA
- BDBDBCADDA
- BBDDBDDCAA
- ... many more

BC … is not valid

Timed SDFG

Single processor schedule using \( q \rightarrow \text{MINIMUM LATENCY!} \)
Throughput Definition

- Actor throughput:
  The \textit{average number of firings} of one actor \textit{per time unit}
  
  \[ \text{Th}(a) = \lim_{k \to \infty} \frac{k \text{ firings of } a}{\text{end time of these firings}}. \]

- (Normalized) graph throughput (if SDFG is consistent):
  \[ \min_{\text{actors } a} \frac{\text{Th}(a)}{q(a)}. \]

Computing throughput for PASS

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

\[ \text{PASS} \rightarrow \text{ACABABCB} \]

\[ \text{Th}(A)=\frac{3}{2+1+2+1+2+1+1+1}=\frac{3}{3\times2+1\times3+1\times3}=\frac{3}{12} \]

\[ \text{Th}(B)=\frac{3}{12}, \text{Th}(C)=\frac{2}{12} \]

\[ \text{Th}(SDG)=\frac{1}{12} \]

Single processor schedule using \( q \rightarrow \text{MINIMUM LATENCY!} \)
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 (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
In this simple-minded approach, the schedule
BBBCDDDDAA
would produce code like

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

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;
<table>
<thead>
<tr>
<th>Single-Appearance Schedules</th>
<th>Minimum-Memory Schedules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Often possible to choose a looped schedule in which each block appears exactly once.</td>
<td>Another possible objective.</td>
</tr>
</tbody>
</table>
| Leads to efficient block-structured code  
  - Only requires one copy of each block’s code | Often increases code size (block-generated code). |
| Does not always exist | Static scheduling makes it possible to exactly predict memory requirements. |
| Often requires more buffer space than other schedules | |
Mapping onto MPSoC Platforms

Multiple time-constrained applications

Provide timing guarantees on mapping of each application

Multiprocessor system

Parallel (multi-core) schedules

- Given $P_s$ (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 throughput is the target), taking into account resource constraints
Example

q=[2,3,3], J=1

Big catch… exponential blowup in #nodes is possible!

Unrolling…

J=1, Th=1/4
J=2, Th=2/4
J=n, Th=n/4
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 \( T \):

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

- This is given *unbounded resources*
  - NOTE: if there are no loops \( T \to 0 \)

Example

- HSDF (from SDF)

\[
T = \frac{(1+1+1)}{1} = 3
\]

Polynomial-time computation on HSDF
Achieving the MCM bound

- Can be achieved with a periodic time-triggered schedule (everything is synchronized) by optimal unrolling \( J_{OPT} \)
  - \( J_{OPT} \) can be determined by a transformation [Parhi91]
    - SDF \( \rightarrow \) HSDF
    - Unfold HSDF \( mcm(\text{delays in loops}) \) 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 firing – periodic part
  - Implementation of STS can be tricky (…but)

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

**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

**Actor** (per processor type: execution time, memory usage)

- Edge (storage space source / destination / memory, token size, bandwidth requirement)

Throughput constraint on graph
Problem statement

Find a binding and scheduling of an SDFG onto an MP-SoC that satisfies the throughput constraint.

Throughput analysis

State: (token distribution, execution times firing actors)

throughput $C = 1/2$
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

throughput C = 1/29

throughput C = 1/30
Resource allocation strategy

- Throughput-constrained SDFG
- MP-SoC architecture

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
  \[ \text{cost}(t) = c_1 \cdot l_p(t) + c_2 \cdot l_m(t) + c_3 \cdot l_c(t) \]
<table>
<thead>
<tr>
<th>Static-order scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Order actor firings of an application on a processor</td>
</tr>
<tr>
<td>- List-scheduling algorithm</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time slice allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Provide timing independence between applications</td>
</tr>
<tr>
<td>- Binary search algorithm using fast throughput analysis technique</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>cost</th>
<th>compute intensive</th>
<th>memory intensive</th>
<th>communication intensive</th>
<th>balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0,0</td>
<td>20.22</td>
<td>5.22</td>
<td>7.56</td>
<td>18.56</td>
</tr>
<tr>
<td>0,1,0</td>
<td>18.78</td>
<td>8.00</td>
<td>11.33</td>
<td>23.33</td>
</tr>
<tr>
<td>0,0,1</td>
<td>29.22</td>
<td>7.56</td>
<td>12.89</td>
<td>25.00</td>
</tr>
<tr>
<td>1,1,1</td>
<td>18.44</td>
<td>6.50</td>
<td>10.33</td>
<td>23.56</td>
</tr>
<tr>
<td>0,1,2</td>
<td>24.56</td>
<td>8.00</td>
<td>12.89</td>
<td>30.11</td>
</tr>
</tbody>
</table>

Ip, Im, Ic
- 16.1 throughput computations per SDFG
<table>
<thead>
<tr>
<th><strong>Experimental results</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Application</td>
</tr>
<tr>
<td>▪ 3x H.263 decoders (4 actors)</td>
</tr>
<tr>
<td>▪ 1x MP3 decoder (13 actors)</td>
</tr>
<tr>
<td>▪ Architecture</td>
</tr>
<tr>
<td>▪ 2x2 mesh of tiles</td>
</tr>
<tr>
<td>▪ 2 accelerators, 2 general-purpose processors</td>
</tr>
<tr>
<td>▪ Cost function (2,0,1)</td>
</tr>
<tr>
<td>▪ Focus on processing and communication</td>
</tr>
<tr>
<td>▪ 34 throughput computations</td>
</tr>
<tr>
<td>▪ Run-time 8 minutes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Conclusions</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Resource allocation strategy for SDFGs on MP-SoCs</td>
</tr>
<tr>
<td>▪ Most expressive model-of-computation used so far</td>
</tr>
<tr>
<td>▪ Technique provides timing guarantees</td>
</tr>
<tr>
<td>▪ Cost functions can steer resource allocation</td>
</tr>
<tr>
<td>▪ Experiments show feasibility of the approach</td>
</tr>
</tbody>
</table>
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