Motivation

- DSP representation & modeling
- to expose the concurrency (parallelism)
- closer to the natural description of DSP applications

- Specification: MATLAB, SPW, Cossap, C/C++
- Floating point
  (DSP Canvas)
- Fixed point
- Algorithm transformations

Architectures alternatives:
- ASIC
- Bit parallel
- Bit serial
- Special Purpose
  (Art Designer)
- Retargetable coprocessor
  (Target compiler technologies, Easics)
- DSP processors
  (TI TMS320C54x)
- DSP extensions to RISC
  (Tensilica)
Introduction:


Data flow

Data flow representation of an algorithm:
- is a directed graph
- nodes are computations
- arcs (or edges) are paths over which the data ("samples") travels.

DF shows which computations to perform, not sequence. Sequence is only determined by data dependencies. Hence exposes concurrency.
Data flow (cont.)

Assume infinite stream of input samples.
So nodes perform computations an infinite times.

Node will “fire” (start its computation) when inputs are available.
Node with no inputs can fire anytime.

Numbers indicate the number of samples (tokens) produced, consumed
by one firing.

Nodes will fire when input data is available, called “data-driven”.
Hence it exposes concurrency.

True data flow: overhead for checking the availability of input tokens
is too large.
BUT, synchronous data flow: the number of tokens produced/consumed
is know beforehand (a priori)!
Hence, the scheduling can done a priori, at compile time. Thus there is NO
runtime overhead!

For signal processing applications: the number of tokens produced
& consumed is independent of the data and known beforehand
(= relative sample rates).
Synchronous Data Flow - definition

Synchronous data flow graph (SDF) is a network of synchronous nodes (also called blocks).
A node is a function that is invoked whenever there are enough inputs available. The inputs are consumed.
For a synchronous node, the consumptions and productions are known a priori.

![SDF Graph]

Homogeneous SDF graph: when only “1”’s on the graph.

Precedence graph - Schedule

Precedence graph indicates the sequence of operations:

Schedule determines when and where (which processor or which data path unit) the node fires.

Valid schedules: 

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

Invalid schedule: 

| C | A | B |
Blocked Schedule

Precedence graph indicates the sequence of operations:

![Graph of operations A, B, C, D, E, F, G with precedence relationships]

Static schedule

3 processors/units: valid blocked schedule

With pipeline:

<table>
<thead>
<tr>
<th>P1</th>
<th>A</th>
<th>C</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>B</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>E</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Small – large grain

Iteration period = length of one cycle = 1/throughput

Goal: minimize iteration period

Iteration period bound = ?

Atomic SDF graph, when nodes are primitive operations

Large grain SDF graph, when nodes are larger functions:

Example: LPS speech processing (will be project)
SDF graph implementation

Implementation requires:
- buffering of the data samples passing between nodes
- schedule nodes when inputs are available

Dynamic implementation (= runtime) requires
- runtime scheduler checks when inputs are available and
  schedules nodes when a processor is free.
- usually expensive because overhead

Contribution of Lee-87:
- SDF graphs can be scheduled at compile time
- no overhead

Compiler will:
- determine the execution order of the nodes on
  one or multiple processors or data path units
- determine communication buffers between nodes.

Periodic schedule for SDF graph

Assumptions:
- infinite stream of input data (the case for signal processing applications)
- periodic schedule: same schedule applied repetitively on input stream

Goal:
- check if schedule can be found:
  - Periodic admissible sequential schedule (PASS) for a single processor or data path unit
  - Periodic admissible parallel schedule (PAPS) for multiple processors

Rate inconsistency

Consistent solution
Formal approach

Construct topology matrix
• each node is a column
• each arc is a row
• entry \((i,j)\) = data produced on node \(i\) by arc \(j\).
• consumption is negative entry

\[
\Gamma = \begin{bmatrix}
e_1 & n_1 & n_2 & n_3 \\
e_2 & 1 & -1 & 0 \\
e_3 & 2 & 0 & -1 \\
& 0 & 2 & -1 \\
\end{bmatrix}
\]

Self loop entry?

FIFO queues

\(b(n) = \text{size of queues on each arc}\)
\(v(n) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ indicates firing node}\)
\(b(n+1) = b(n) + \Gamma v(n)\)

\[
\begin{bmatrix}
e_1 & n_1 & n_2 & n_3 \\
e_2 & 1 & -1 & 0 \\
e_3 & 2 & 0 & -1 \\
& 0 & 2 & -1 \\
\end{bmatrix}
\]

\[
b(0) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad b(1) = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}
\]
FIFO queues & delays

Delays are handled by initializing $b(0)$ with the delay values:

$$b(0) = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

So at start-up:
- can fire $n_3$ two times before firing $n_1$ again

Identifying inconsistent sample rates

**Necessary** condition for the existence of periodic schedule:

Rank of $\Gamma$ is $s-1$ ($s$ is number of nodes)
Relative firing frequency

Topology matrix with the correct rank, has a strictly positive (element-wise) integer vector $q$ in its right nullspace:

Thus: $\Gamma q = 0$

$q$ determines number of times each node is invoked!

Insufficient delays

Rank $s-1$ is a necessary but not a sufficient condition:

$$\begin{bmatrix} 1 & -1 & 1 \\ -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$
Scheduling for single processor

Given:
- positive integer vector $q$, such that $\Gamma q = 0$
- given $b(0)$
The i-th node is “runnable” if
- it has not been run $q_i$ times
- it will not cause the buffer size to become negative

Class “S” (sequential) algorithm:
- is an algorithm that schedules a node if it is runnable
- it stops when no more nodes are runnable.

If the class S algorithm terminates before it has scheduled each node the number of times specified in the $q$ vector, then it is said to be deadlocked.

Example Class S algorithm

- Solve for smallest positive integer $q$
- Form a list of all nodes in the system
- for each node, schedule if runnable, try each node once
- if each node has been scheduled $q_i$ times, STOP.
- If no node can be scheduled, indicate deadlock
- else continue with the next node.

Schedule:
- 1 - 2 - 3 - 3 is PASS
- 1 - 2 - 3 is not PASS
- 2 - 1 - 3 - 3 is not PASS

(Complexity: traverse the graph once, visiting each edge once).
Optimization: minimize buffer (=memory) requirements
Schedule for parallel processors

Assumptions:
• homogeneous processors, no overhead in communication
• if PASS exists, then also PAPS
  (because we could run all nodes on one processor)

A blocked periodic admissible parallel schedule is
• set of lists \( \{X_i; i = 1, \ldots, M\} \)
• \( M \) is the number of processors
• \( X_i \) = periodic schedule for processor \( i \)

\( p \) is smallest positive integer vector, such that \( \Gamma p = 0 \).
Then a cycle of schedule invokes every node \( q = Jp \) times.
\( J \) is called the blocking factor (can be different from 1).

Precedence graph

Assumptions:
\[ \begin{array}{c}
| n1 & e1 & n2 & e2 & n3 & e3 \\
1 & 1 & 2D & 2 & 1 & 2D \\
\end{array} \]
\[ \begin{array}{c}
| n1 & n2 & n3 \\
1 & -2 & 0 \\
0 & 1 & -1 \\
-1 & 0 & 2 \\
\end{array} \]
\[ \begin{array}{c}
\text{rank} = 2, \ p = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} \\
\Gamma p = 0. \\
\text{PASS: ?} \\
\end{array} \]

Precedence graph for unity blocking factor:
Schedule on two processors, J=1

Assumptions:
- node 1 takes 1 time unit, node 2 takes 2, node 3 takes 3

- $X_1 = \{3\}$
- $X_2 = \{1, 1, 2\}$

Schedule on two processors, J=2

Assumptions:
- node 1 takes 1 time unit, node 2 takes 2, node 3 takes 3
- nodes have self loops (so nodes can not overlap with themselves)

- $X_1 = \{3, 1, 1, 2\}$
- $X_2 = \{1, 1, 2, 3\}$
Why are we doing this?

The principle of synchronous data flow is used in many simulators:
OcapI - SPW

Will model our speech processing project in this way.
Combine with bit true simulations and real-life is closely modeled.

Issues in practice:
• choose schedule to minimize memory requirements.
• include non data flow nodes
  • if-then-else
  • data dependent calculations