Data Flow on a Queue Machine

Bruno R. Preiss
Outline

- Genesis of data-flow architectures
- Static vs. dynamic data-flow architectures
- Pseudo-static data-flow execution model
- Some data-flow machines
- Simple queue machine
- Prioritized queue machine
- Program decomposition
- Queue machine processing element
Genesis of Data Flow

- Data-flow principles: asynchrony and functionality

\[
x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}; \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}
\]
Data-Flow Execution Models

Static:
- Program loaded into memory in completed form before execution begins
- At most one instance of an actor can be enabled
- Same storage space used for instructions and data

Dynamic:
- Program nodes can be instantiated at run time
- Several instances of an actor may be enabled
- Separate storage space used for instructions and data
Pseudo-static Data Flow

- Can associate several data spaces with one instruction space
- Reentrancy accomplished without code copying or tagged tokens
MIT Static Data-Flow Machine

- Static data-flow machine
- Packet communication based
- Networks are pipelined — provide queueing
- 4 processor prototype built

LAU System Architecture

- Static data-flow machine
- Uses acyclic data-flow graphs
• Explicit external instruction queue
• 32 processor prototype built
Manchester Data-Flow Architecture

- Dynamic data-flow architecture
- Tagged tokens
- Circular pipeline with token queue
- 15 processor prototype built

MIT Dynamic Data-Flow Architecture

- Dynamic data-flow architecture
• Tagged tokens
• Circular pipeline with queueing in waiting/matching section
Simple Queue Machine

- Uses a first-in, first-out (FIFO) queue for the manipulation of operands and results
- Its instructions implicitly reference an operand queue
- Analogous to a stack machine

\[ f \leftarrow ab + \frac{(c-d)}{e} \]

<table>
<thead>
<tr>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction sequence</td>
<td>stack contents</td>
</tr>
<tr>
<td>fetch a</td>
<td>a</td>
</tr>
<tr>
<td>fetch b</td>
<td>b, a</td>
</tr>
<tr>
<td>mul</td>
<td>ab</td>
</tr>
<tr>
<td>fetch c</td>
<td>c, ab</td>
</tr>
<tr>
<td>fetch d</td>
<td>d, c, ab</td>
</tr>
<tr>
<td>sub</td>
<td>c–d, ab</td>
</tr>
<tr>
<td>fetch e</td>
<td>e, c–d, ab</td>
</tr>
<tr>
<td>div</td>
<td>(\frac{(c-d)}{e}, ab)</td>
</tr>
<tr>
<td>add</td>
<td>(ab + \frac{(c-d)}{e})</td>
</tr>
<tr>
<td>store f</td>
<td>store f</td>
</tr>
</tbody>
</table>
Generating Instruction Sequences
for a Simple Queue Machine

\[ f \leftarrow ab + \frac{c-d}{d} \]

- LEMMA: Level-order traversal of expression parse tree gives queue machine instruction sequence
- Time complexity of level order conjugation is \( O(N) \)
- Space complexity of level order conjugation is \( O(N) \)
Pipelined Execution: Stack vs. Queue

- Case 1: Non-overlapped operand fetch/execute
- Case 2: Overlapped operand fetch/execute

\[
Speed-up = \frac{Stack\ Machine\ Cycles}{Queue\ Machine\ Cycles}
\]

<table>
<thead>
<tr>
<th>nodes in parse tree</th>
<th>number of trees</th>
<th>case 1</th>
<th>case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>1.</td>
<td>1.</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>1.02</td>
<td>1.02</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>1.03</td>
<td>1.03</td>
</tr>
<tr>
<td>7</td>
<td>45</td>
<td>1.04</td>
<td>1.05</td>
</tr>
<tr>
<td>8</td>
<td>101</td>
<td>1.05</td>
<td>1.07</td>
</tr>
<tr>
<td>9</td>
<td>227</td>
<td>1.05</td>
<td>1.08</td>
</tr>
<tr>
<td>10</td>
<td>510</td>
<td>1.05</td>
<td>1.09</td>
</tr>
<tr>
<td>11</td>
<td>1146</td>
<td>1.06</td>
<td>1.10</td>
</tr>
</tbody>
</table>

\[\uparrow\text{two-stage pipelined ALU}\]

<table>
<thead>
<tr>
<th>pipeline stages</th>
<th>assumption 1</th>
<th>assumption 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.</td>
<td>1.</td>
</tr>
<tr>
<td>2</td>
<td>1.06</td>
<td>1.10</td>
</tr>
<tr>
<td>3</td>
<td>1.08</td>
<td>1.09</td>
</tr>
<tr>
<td>4</td>
<td>1.09</td>
<td>1.08</td>
</tr>
<tr>
<td>5</td>
<td>1.10</td>
<td>1.07</td>
</tr>
</tbody>
</table>

\[\uparrow11\ nodes\ in\ parse\ tree\]
Prioritized Queue Machine

- Assign a “priority” to the result of each operation
- Place result in queue at position determined from priority

\[
\begin{align*}
d & \leftarrow \frac{a}{a+b} \cdot (a+b) c
\end{align*}
\]

Prioritized Queue Machine

<table>
<thead>
<tr>
<th>Instruction Sequence</th>
<th>Result Priorities</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch (a)</td>
<td>0, 2</td>
<td>(a, \varepsilon, a)</td>
</tr>
<tr>
<td>fetch (b)</td>
<td>1</td>
<td>(a, b, a)</td>
</tr>
<tr>
<td>plus</td>
<td>1, 2</td>
<td>(a, a+b, a+b)</td>
</tr>
<tr>
<td>fetch (c)</td>
<td>3</td>
<td>(a, a+b, a+b, c)</td>
</tr>
<tr>
<td>div</td>
<td>2</td>
<td>(a+b, c, \frac{a}{a+b})</td>
</tr>
<tr>
<td>mul</td>
<td>1</td>
<td>(\frac{a}{a+b}, (a+b)c)</td>
</tr>
<tr>
<td>add</td>
<td>0</td>
<td>(\frac{a}{a+b}, (a+b)c)</td>
</tr>
</tbody>
</table>

store \(d\)

- LEMMA: Acyclic data-flow graphs “generate” valid prioritized queue machine instruction sequences
Dynamic Data-Flow Graph Splicing

- Based on two concepts: *channels* and *contexts*
- Channel: Unidirectional communication path between two contexts
- Context: A process that evaluates an acyclic data-flow graph
- State of a context: An instruction sequence (and PC) and an operand queue
- Conditional execution, iteration, subroutine calls implemented by instructions for context creation (fork) and intercontext communication (send (!) and receive (?))
Partitioning Programs

- Goal: exploit potential parallelism
- Use acyclic data-flow graph as the basic granule of computation
- Granule size trade-off:
  Small contexts → excessive intercontext communication and context generation overhead  
  Large contexts → cannot exploit intracontext parallelism
- Context-based partition is a compromise between conventional data-flow and task- or process-based parallelism (Ada, Concurrent Euclid)
- Conventional data-flow architectures attempt to detect and exploit operator-level parallelism at execution time (costly and inefficient)
- Task- or process-based approach requires programmer to explicitly partition programs into granules of computation and to code communications between tasks
- Approach: Partition programs (automatically) into granules of computation more complex than single instructions, yet less complex than processor task, and to automatically exploit parallelism between contexts
- OCCAM compiler
Queue Machine Processing Element

- QP: queue pointer register
- Allocate page of memory for queue of each context
- Use window registers as a queue cache
- Presence bits indicate validity of register contents
Queue Machine Processing Element Architecture
Queue Machine Assembly Language

- Syntax: opcode{+[src1[,src2]][+dst1[,dst2]]}

\[ d \leftarrow \frac{a}{a+b} + (a+b)c \]

```
fetch #a: r0, r2
fetch #b: rf
plus++ r0,r1: r1, r2
fetch #c: f3
div++ r0,r1: r2
mul++ r0,r1: r1
plus++ r0,r1: r0
store+ #d,r0
```