Dataflow Programming

By James Kempsell & Steve Smith
Outline

● Motivation
● History
● Architectures
  ○ Static Dataflow Architecture
  ○ Manchester Prototype Dataflow Machine
● High Level Dataflow Language: CAL
● Dataflow issues and Future
● Conclusion
Motivation For Dataflow

- Control-flow performance is largely based on the program's
  1. Instruction-Level Parallelism
  2. Data-Level Parallelism
  3. Thread-Level Parallelism

- These often restrict the concurrency of a program
Motivation for Dataflow

- Dataflow allows an instruction to run as soon as all of its operands are ready.

- Instructions do not impose any constraints on sequencing except for the data dependencies.

- Synchronization of parallel activities is implicit in the dataflow model.
History

- The concept of dataflow programming was created in the early 1970's.

- Development of effective software and hardware architectures continued until around the late 1980's.
History

- Dataflow program lost interest and support overtime because:
  - Dataflow had more overhead per instruction cycle compared to control-flow
  - Detection of enabled instructions and generation of result tokens lead to poor performance in applications with a low degree of parallelism
  - Since the execution of an instruction has to consume input tokens and generate output tokens, extra communication of tokens among instructions is required
Dataflow Architectures

1. Static

Organization of the Static Dataflow architecture model
Static Model

- **Memory**
  - Stores instructions in memory cells

- **Processing**
  - Has five pipelined functional units

- **Network**
  - **Arbitration**
    - Send enabled instructions from memory to processing
  - **Distribution**
    - Send results from processing to memory
  - **Control**
    - Transfers tokens and acknowledgement signals from processing to memory
Dataflow Architectures

Dynamic Tagging: Manchester Dataflow

1. Token Queue
2. Matching Unit
   a. Matches Tokens that go to the same node
3. Instruction Store
   a. Only memory access is to get relevant instructions
4. Processing Unit
   a. Output tokens go back to queue
Example: Numerical Integration

for initial
    int := 0.01
    Y := 0.0;
    X := 0.02
while
    x < 1.0
repeat
    int := 0.01 * (old y + y);
    Y := old x * old x;
    x := old x + 0.02
returns
    value of sum int
end for

• Only allowed one write per variable
• Explicitly declares this loop as parallel
• "Type 3" Parallelization
• "old x" and "old y" are the previous iteration's values
Manchester Instruction Meanings

- **DUP**-Duplicate the signal. Needed because max outputs is 2.
- **ADR**-add floating-point values.
- **BRR**-branch. This has 2 inputs (condition, and thread) and 2 outputs (options to go next).
- **CGR**-compare floating point values.
- **MLR**-multiply floating-point values.
- **OPT**-send output to host processor.
- **ADL**-add to iteration level.
- **SIL**-set iteration level.
- First x input compared with 1.0
- Signal duplicated to all branches
- If true, continue executing
- Else, send output

ADR-add floating-point values
BRR-branch
CGR-compare floating point l.h. > r.h.
ADL-add to iteration level
MLR-multiply floating-point values
OPT-send output to host processor
SIL-set iteration level
DUP-Duplicate Signal
- Duplicate input x
- Multiply by itself (square)
- Send to current iteration as y
- Send to next iteration as old y

ADR-add floating-point values
BRR-branch
CGR-compare floating point l.h. > r.h.
ADL-add to iteration level
MLR-multiply floating-point values
OPT-send output to host processor
SIL-set iteration level
DUP-Duplicate Signal
- Add old y + y
- Multiply by .01
- Accumulate integral

ADR - add floating-point values
BRR - branch
CGR - compare floating point l.h. > r.h.
ADL - add to iteration level
MLR - multiply floating-point values
OPT - send output to host processor
SIL - set iteration level
DUP - Duplicate Signal

FIGURE 1. Dataflow Graph for the Integration Program
### Dataflow Architectures

- Run with sufficiently large data sets
- Hardware naturally parallelizes the code, giving over 90% efficiency with 12 units
- Measured from first input, to first output on real hardware, not simulation

<table>
<thead>
<tr>
<th>Function units (n)</th>
<th>Run time (seconds) (Tn)</th>
<th>Speedup (Pn)</th>
<th>Efficiency (En)</th>
<th>Actual MIPS (Mn)</th>
<th>Potential MIPS (Mn')</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.4215</td>
<td>1.00</td>
<td>100.0</td>
<td>0.117</td>
<td>0.117</td>
</tr>
<tr>
<td>2</td>
<td>2.2106</td>
<td>2.00</td>
<td>100.0</td>
<td>0.235</td>
<td>0.235</td>
</tr>
<tr>
<td>3</td>
<td>1.4751</td>
<td>3.00</td>
<td>99.9</td>
<td>0.352</td>
<td>0.352</td>
</tr>
<tr>
<td>4</td>
<td>1.1077</td>
<td>3.99</td>
<td>99.8</td>
<td>0.469</td>
<td>0.470</td>
</tr>
<tr>
<td>5</td>
<td>0.8886</td>
<td>4.98</td>
<td>99.5</td>
<td>0.585</td>
<td>0.587</td>
</tr>
<tr>
<td>6</td>
<td>0.7429</td>
<td>5.95</td>
<td>99.2</td>
<td>0.699</td>
<td>0.705</td>
</tr>
<tr>
<td>7</td>
<td>0.6400</td>
<td>6.91</td>
<td>98.7</td>
<td>0.812</td>
<td>0.822</td>
</tr>
<tr>
<td>8</td>
<td>0.5643</td>
<td>7.84</td>
<td>97.9</td>
<td>0.921</td>
<td>0.940</td>
</tr>
<tr>
<td>9</td>
<td>0.5071</td>
<td>8.72</td>
<td>96.9</td>
<td>1.024</td>
<td>1.057</td>
</tr>
<tr>
<td>10</td>
<td>0.4629</td>
<td>9.55</td>
<td>95.5</td>
<td>1.122</td>
<td>1.175</td>
</tr>
<tr>
<td>11</td>
<td>0.4301</td>
<td>10.28</td>
<td>93.5</td>
<td>1.208</td>
<td>1.292</td>
</tr>
<tr>
<td>12</td>
<td>0.4038</td>
<td>10.95</td>
<td>91.3</td>
<td>1.287</td>
<td>1.410</td>
</tr>
</tbody>
</table>
Dataflow Architectures

3. Explicit Token Store

Explicit-token-store representation of a dataflow program
High Level Dataflow Language: CAL

- Cal Actor Language
- Developed by UC Berkeley in 2003, aimed at multiple platform support, and a focus on their Ptolemy II platform
- Program each node in a flow graph as an "actor"
- Actor has input, output and internal state
Basic Actor: Buffer

actor Buffer () In ==> Out :
   action In: [a] ==> Out: [a] end
end

● Declares actor "Buffer" with no parameters, and 1 input port: In and one output port: Out
● 1 action takes a token from In, names it a, and sends it to out
● availability of "a" defines when this actor's action will fire
● A single actor may have multiple actions
Actor State

actor Sum () Input ==> Output:
sum := 0;
action [a] ==> [sum] do
  sum := sum + a;
end
end

- Sum initialized to zero, then increases inside the action
  - similar syntax to VHDL
- Can use the "old" keyword like SISAL
- Internal state can define where output goes, what output is needed
- allows easy construction of complex state machines
Some Remaining Issues

- There is no concept of a variable when using dataflow programming tokens
  - Handling data structures like arrays become difficult
  - Might be solved with "direct access" or "indirect access"

- There is a need for a better method for program allocation
  - Maximize parallelism, minimize communication
Some Remaining Issues

● Memory Latencies account for a lot of wasted execution time
  ○ Each instruction is a thread, and dataflow programming does not allow context switching (each instruction runs to completion once it’s started)
  ○ One proposed solution is to use a cache memory

● Not intuitive Programming
  ○ Sequential programming is intuitive
  ○ Parallel programming typically has chunks of intuitive sequential computation (grain size)
  ○ Dataflow is 100% parallel
Conclusion

● Should Dataflow programming be used right now?
  ○ No, not by itself. There are still some issues that make control-flow programming more effective in parallel models
  ○ Hybrid solutions are viable now

● Is it worth it to continue research of Dataflow programming?
  ○ Yes, it has the potential to become the best solution for parallel systems
Questions or Comments?