Data Flow on a Queue Machine

by

Bruno R. Preiss
Goal

• to develop a realistic and efficient multiprocessor computer architecture based on the data-flow execution model

Approach

• use elements of conventional Von Neumann architectures to supplant aspects of the data-flow execution model

Thesis

• the goal can be achieved using a multiprocessor system based on the pseudo-static data-flow execution model
• in this model, a computation is partitioned into a collection of acyclic data-flow graphs
• a natural execution mechanism for evaluating acyclic data-flow graphs is the indexed queue machine
• an indexed queue machine processing element can operate efficiently since it can take advantage of pipelined execution and operand caching
Overview of Research

I Theory
- develop mathematical foundation for queue-based execution models
- queue based execution models:
  - simple queue machine
  - indexed queue machine
- queue machines vs stack machines

II Programming Issues
- compiler design issues associated with generating queue machine code
- OCCAM compiler implementation:
  - partitioning rules
  - code generation algorithms
  - heuristic optimizations

III Architectural Issues
- design of processing element/multiprocessor architecture
- processing element architecture:
  - window registers
  - virtual register numbers
- multiprocessor architecture:
  - based on P-bus
IV Simulation/Performances Issues

- characterize performance
- execute programs generated by OCCAM compiler
- register-transfer-level simulation
- kernel implementation
Data Flow Execution Models

I Static
- program loaded into memory in completed form before execution
- at most one instance of an actor can be enabled
- same storage space used for instructions and data
- pros: simple firing rule evaluation
- cons: inefficient function calls, loop unravelling impossible, reentrancy impossible

II Dynamic
- program nodes can be instantiated at run time
- several instances of an actor may be enabled
- separate storage spaces for instructions and data
- pros: function calls, loop unravelling, reentrancy all possible
- cons: tag manipulation overhead
### III Pseudo-static

- dynamically allocate blocks of memory for data at run time
- one block per ‘‘context’’
- statically allocate within such blocks
- reentrancy accomplished without overhead of tagged tokens or code copying

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand Slot</th>
<th>Context A</th>
<th>Data Space</th>
<th>Context B</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0</td>
<td>operand slot</td>
<td>a, b, c, d</td>
<td>a', b', c', d'</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>operand slot</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td>2</td>
<td>operand slot</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>operand slot</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div</td>
<td>4</td>
<td>operand slot</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>operand slot</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright (c) 1987 by Bruno R. Preiss, P.Eng. All rights reserved.
Queue Machines

Simple Queue Machine

- uses FIFO queue for manipulation of operands and results
- instructions implicitly reference an operand queue
- analogous to a stack machine
- Lemma: Evaluation of level-order traversal of an expression parse tree on a simple queue machine correctly computes the expression
- Lemma: In-order traversal of level-order conjugate of a binary tree is identical to level-order traversal
- Algorithm: construct level-order conjugate in O(n) time and space

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

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

- assign an index to the result of each operation
- place result in queue at position specified by the index
- Lemma: Evaluation of valid indexed queue machine instruction sequence for a given acyclic data-flow graph correctly computes the expression(s) represented by that graph
- indexed queue machine is the natural execution mechanism for evaluating acyclic data-flow graphs

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result Indices</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fetch a</code></td>
<td>0,2</td>
<td><code>a, ε, a</code></td>
</tr>
<tr>
<td><code>fetch b</code></td>
<td>1</td>
<td><code>a, b, a</code></td>
</tr>
<tr>
<td><code>add</code></td>
<td>0</td>
<td><code>a/(a+b)+(a+b)c</code></td>
</tr>
<tr>
<td><code>div</code></td>
<td>2</td>
<td><code>a+b, c, a/(a+b)</code></td>
</tr>
<tr>
<td><code>mul</code></td>
<td>1</td>
<td><code>a/(a+b), (a+b)c</code></td>
</tr>
<tr>
<td><code>store</code></td>
<td></td>
<td><code>d</code></td>
</tr>
</tbody>
</table>
Programming Issues

- goal: to exploit potential parallelism
- approach: use acyclic data-flow graph as basic granule of computation
- granularity trade-off:
  - small contexts $\rightarrow$ excessive intercontext communication and context generation overhead
  - large contexts $\rightarrow$ cannot exploit intracontext parallelism
- context-based partition: compromise between conventional data flow and task- or process-based parallelism
- conventional data flow: attempt to detect operator-level parallelism at execution time (costly and inefficient)
- task- or process-based approach: requires explicit programmer action to partition and coordinate parallel tasks
- approach: partition programs (automatically) into granules of computation more complex than single instructions, yet less complex than processes or tasks, and exploit parallelism between contexts
OCCAM Compiler

I Partitioning Rules
   • create separate contexts for:
     • each procedure body
     • each while loop body
     • each replicated par or seq construct
     • each conditional outcome

II Graph Generation Algorithm
   • based on data-flow analysis procedure of Allan and Oldeheuft
   • top-down, recursive descent flow analysis procedure
   • table based: I set, O set, E set, and T tree

III Input Arc Sequencing
   • heuristic
   • attempt to maximize parallelism by maximizing amount of computation possible in a context before it blocks
   • sequence inputs based on cost of computation

IV Actor Sequencing
   • heuristic
   • data-flow graph does not uniquely specify indexed queue machine instruction sequence
   • attempt to maximize parallelism
• (ignore other factors, e.g., register utilization efficiency)
Architectural Issues

Queue Machine Processing Element

- allocate a memory page for the queue of each context
- QP: queue pointer register
- window registers act as queue cache
  - presence bits indicate validity of register contents
- virtual-register-number-based instruction set
Architectural Issues

P-bus-based Multiprocessor Architecture

- partitioned, shared-bus architecture
- shared memory used for global data
- local memory used for program and queue data
- dedicated message cache hardware used for intercontext communications via OCCAM-like channels

![Diagram of P-bus-based Multiprocessor Architecture]
Performance Analysis

Amdahl’s Law

- $R_n$: system throughput ratio: ratio of throughput on an $n$ processor system to single processor system
- $X$: workload (assumption: independent of $n$)
- $T_n = \frac{P}{n} + S$: total execution time on an $n$ processor system

$$R_n = \frac{X/T_n}{X/T_1} = \frac{T_1}{T_n},$$

$$R_n = \frac{1}{1 + (1/n - 1)f}, \quad f = \frac{P}{P+S}$$

$$\lim_{n \to \infty} R_n = \frac{1}{1-f}$$

$$\left. \frac{dR_n}{dn} \right|_{n=1} = f$$

frame invis ht 2 wid 4 left solid bot solid coord x 1.40 y 1.20 ticks top at 1 "R_n" ticks right at 1 "n" ticks left at 1, 8 ticks left out .05 from 2 to 7 by 1 "" ticks bot at 1, 8 ticks bot out .05 from 2 to 7 by 1 "" tick left at 14.29 "1/" "slope = f" ljust at 18.5,17 define amdahl {
  next at 1.1/(1+(1/1 - 1) * .93 ) } draw dotted 1 1 20 20 new dashed 1 14.29 40 14.29 new dashed 1 1 21.43 20 new solid for i from 1 to 40 by .5 do { amdahl(i) }
Performance Analysis

Modified Amdahl’s law

- $X = X^U + X^K$: total workload
- $T_n = T_n^U + T_n^K$: total execution time
- $T_n^U = \frac{P_U}{n} + S^U$: as before
- $T^K = \frac{Y}{n}$: assumption: independent kernel calls
- $Y = \frac{P^K}{n} + S^K$: kernel workload depends on $n$

$$R_n = \frac{X^U / T_n}{X^U / T_1} = \frac{T_1}{T_n}$$

$$R_n = \frac{1}{1 + (1/n - 1)f + (1/n^2 - 1)g}, \quad f = \frac{S^K + P_U}{P^K + S^K + P_U + S^U}, \quad g = \frac{P^K}{P^K + S^K + P_U + S^U}$$

$$\lim_{n \to \infty} R_n = \frac{1}{1 - (f + g)}$$

$$\frac{dR_n}{dn} \bigg|_{n=1} = f + 2g$$

- $f + 2g$ can be $>1 \rightarrow$ super-linear speed-up frame
Simulation Results

- programs:
  - matrix multiplication
  - fast Fourier transform
  - Cholesky decomposition
  - congruence transformation

- number of processors: 1-8
  - frame invis ht 6 wid 3 left solid bot solid
  - label left "System" "Throughput" "Ratio" left .5 label bot "Number of Processors" coord x 1,8 y 1,16 ticks left at 1 ticks left from 2 to 16 by 2 ticks bot from 1 to 8 "matmul" ljust at 8.1,8.07 "fft" ljust at 8.1,6.48 "cholesky" ljust at 8.1,8.42 "congruence" ljust at 8.1,15.21 new dotted 1 1 8 8 draw A solid draw B solid draw C solid draw D solid copy thru % next A at 1,2 bullet at 1,2 next B at 1,3 bullet at 1,3 next C at 1,4 bullet at 1,4 next D at 1,5 bullet at 1,5 % 1 1 1 1 1 2 2.82 2.74 3.14 3.56 3 4.51 4.21 5.12 6.21 4 5.74 5.33 6.48 9.47 5 6.92 5.68 7.90 12.06 6 7.31 6.30 7.94 13.49 7 7.34 6.46 8.61 13.81 8 8.07 6.48 8.42 15.21
Summary

I Original Contributions
- pseudo-static data-flow execution model
- queue-based execution models:
  - simple queue machine
  - indexed queue machine
  - algorithm for instruction sequence generation
  - complexity proofs
  - queue machine processing element architecture
- cache-based message handling scheme
- modified Amdahl’s law

II What surprised me
- that OCCAM maps easily onto a data-flow machine
- better than linear speed-up performance

III Suggestions for Further Research
- evaluate possible DSP applications of queue machines
- integrate stack- and queue-based machine concepts
- OCCAM compiler issues:
  - improved partitioning algorithms (AI)
  - improved memory allocation
- QMPE control logic design
- message handling hardware issues
• operating systems issues:
  • virtual memory support
  • multi-user support