Dataflow Architectures

Karin Strauss
Introduction

• Dataflow machines: *programmable* computers with hardware optimized for *fine grain data-driven* parallel computation

• *fine grain*: at the instruction granularity

• *data-driven*: activated by data availability
  ▪ only data dependences constrain parallelism
  ▪ programs usually represented as graphs
Terminology

- nodes (FUs)
- tokens (data)
- arcs (dependences)
- input port
- enabling
- firing
- input data + firing = output data
Node Types

- **functional**: (e.g. +, -, * ...)

- **conditional**:
  - value if control == true
  - nothing if control == false

- **merge**:  
  - input 1
  - input 2
  - non-strict firing rule
  - acts as a serializer
if (u < v)
  z = x + y;
else
  z = x - y;
Iteration, Recursion, Reuse and Reentrance

- using the *same graph* to perform computation on *different data sets*

- assume no storage elements are used, data is only present in wires
suppose this node takes 10x what a regular node takes to perform computation.
Synchronization

- **static**
  - locks (compound branch and merge nodes)
    - nodes only fire when all inputs are ready
    - loss of concurrency
  - acknowledging (control flow protocol)
    - extra arcs from consumer to producer
    - increases resources needed
      - can be reduced with detailed analysis
Synchronization II

- dynamic
  - each iteration is executed in a separate instance of the graph
  - code copying
    - new instance of subgraph is created per iteration
    - need to direct tokens from earlier iterations to inputs of new iteration
  - tagged tokens
    - attach a tag to each token, associating it with an iteration
    - fire when input tokens have all the same tag
Tagged Tokens

• create other problems
  ▪ how to manage tags (size, distribution)
  ▪ storage overhead
    • tags have to be stored with tokens
    • tokens that cannot be consumed at the moment may need to be stored for later use
  ▪ too much parallelism!
    • storage overflow
    • deadlocks
Procedures

- procedures can be called from several distinct calling sites
  - callee node address sent in special token
  - mechanism to direct the “return value” token
• several processing elements (PEs) that communicate with each other

1) enabling unit receives token

2) enabling unit stores token at addressed node

3) if node is now enabled, send node to functional unit

4) functional unit processes node

5) output + destination address are sent back to enabling unit
Tagged Architectures

1) matching unit receives token
2) check memory; if all other inputs with same tag are there, send all tokens to fetching unit
3) fetching unit retrieves node from memory
4) fetch unit assembles an executable packet and sends it to functional unit
5) functional unit executes node with inputs provided by packet
6) output is sent back to matching unit
One-level Architecture

- a functional unit delivers tokens to the enabling unit of the correct processing element
Two-level Architecture

- each functional unit consists of several functional elements that can process packets in parallel
Two-stage Architecture

- each enabling unit can send executable packets to any functional unit
  - good for heterogeneous functional units
Manchester Dataflow Machine

- Gurd and Watson (1976-1981), two-level machine

- Pipelined ring
- Matching unit pairs tokens
- Large data sets overflow to overflow units
- Appropriate instruction is fetched from instruction store
- Inputs and instruction are forwarded to processing unit
Underutilization

- poor performance
- underutilization of functional units
  - imbalance
  - overhead computation
- underutilization of storage space
  - large data sets
  - code replication
  - tags, destination addresses
Dataflow Model

• Benefits:
  ▪ exposes parallelism
  ▪ can tolerate latencies
  ▪ mechanisms for fine-grain synchronization

• Drawbacks:
  ▪ loss of locality (interleaving of instructions)
  ▪ waste of resources
  ▪ space overhead
Trends

- convergence of dataflow architectures with conventional Von-Neumann architectures
- Ianucci’s hybrid approach
- decoupled architectures
- out-of-order processors
Ianucci’s Hybrid Approach

- scheduling quanta
  - little or no parallelism among instructions
  - maximize run length (more locality)
  - minimize # of arcs between quanta
  - increase resource utilization
  - deadlock avoidance: dependence sets
Ianucci’s Hybrid Approach (II)
Decoupled Architectures and Out-of-Order Processors

- Decoupled architectures:
  - decentralized structures (distinct FUs)
  - instruction steering based on input and output dependences, and operation to be performed

- Out-of-Order processors:
  - register renaming to identify “iteration”
  - instruction scheduling based on ready input operands
  - predication?
Less Traditional Proposals

• Wavescalar (U. Washington, Mark Oskin)
  ▪ PIM architecture
  ▪ supports traditional Von-Neumann style memory semantics in a dataflow model
  ▪ any programming language

• Edge/Trips (U.T. Austin, Doug Burger)
  ▪ direct instruction communication (within blocks)
  ▪ groups of 128 instructions mapped to an array of execution units: dataflow within, sequential across
  ▪ loads and stores still do through memory ordering hw
Related OoO Techniques

- instruction collapsing (Micro’37)
  - strands: dependent instructions with intermediate computation that does not need to be committed to architectural state (Wills - Georgia Tech)
    - e.g. \(e = a + (b + (c + d))\)
  - mini-graphs: sequence of instructions with at most 2 inputs, 1 output, one memory operation and 1 terminal control transfer (Roth - U. Pennsylvania)
    - goal: save processor resources
      - instruction queue entries
      - reorder buffer entries
      - registers / register file accesses