Dataflow Architectures

- Overview
- Static dataflow
- Dynamic dataflow
- Explicit tag store
- Recent Case Study: Wavescalar

Overview

- A global version of Tomasulo's algorithm (Tomasulo's alg came first)
- Example:
  ```
  input a, b
  y := (a+b)/x
  x := (a*(a+b)) + b
  output y, x
  ```
  note ordering of statements in program is irrelevant

Loop Example

```plaintext
input y, x
n := 0
while y < x do
    y := y + x
    n := n + 1
end
output y, n
```
(Using arrays was intentionally avoided)

Static Dataflow

- Combine control and data into a template
  - like a reservation station
  - except they are held in memory
  - can inhibit parallelism among loop iterations
  - re-use of template ⇒ acks

Dynamic Dataflow

- Separate data tokens and control
  - Token: labeled packet of information
- Allows multiple iterations to be simultaneously active
  - shared control (instruction)
  - separate data tokens
  - A data token can carry a loop iteration number
- Match tokens' tags in matching store via assoc. search
  - if match not found, make entry, wait for partner
  - When there is a match, fetch corresponding instruction from program memory
- Requires large associative search
  - to match tokens
- Adds "structure storage"
  - access via select function – index and structure descriptor as inputs
Dataflow: Advantages/Disadvantages

**Advantages:**
- no program counter
data-driven
- execution inhibited only by true data dependences
- stateless / side-effect free
  further enhances parallelism

**Disadvantages**
- no program counter
  leads to very long fetch/execute latency
- spatial locality in i-fetch hard to exploit
- requires matching (e.g., via associative compares)
- no shared data structures
  no pointers into data structures (implies state)
  in theory take entire data structure as input "token" and emit a
  new version
- I/O difficult - depends on state
  Virtual memory??

Explicit Token Store

- Used in Monsoon
- Implement matching store with RAM
- Frame is associated with an instruction block
  e.g. loop iteration
- Tag contains instruction pointer (IP) and frame pointer (FP)
- Instructions include frame offset to give location for token "rendezvous"
  An instruction indicates destination instructions
  - when a token arrives at instruction unit, instruction is fetched,
  - instruction indicates frame offset for token rendezvous
  - frame location is full/empty
    - full => match, feed both tokens to instruction for execution

Hybrid Solution

- Used in "T"
- Use dataflow at high level
  Dataflow functions are coarser grain operations
- Use conventional control flow to implement coarse grain operations

Wavescalar

- Recent research project at the other UW
  Goal: to minimize communication costs
  - transistors get faster faster than wires get faster
- Keep von Neumann memory model
  - State + side effects
  - "solves" data structure problem (but doesn't get the side effect free advantage)
  - Memory ordering must be explicitly expressed
- Control still follows dataflow model
  - Essentially what is done with Tomasulo's algorithm, but on a much larger scale
  - Attempts to exploit "dataflow locality" in instruction stream
  - Predictability of data dependences
  - Avoid register file/bypass network complexity
  Example: 92% of inputs come from one of the last two producers of the input
  Claim: register renaming destroys locality

Basic Concepts

- Intelligent instruction words
  - Each instruction in memory has its own functional unit (in theory)
- In practice
  - Use WaveCaches near functional units to execute working set of instructions near the functional units
- Control flow broken into "waves"
  - Maximal loop-free section of dataflow graph
  - Single entry point
  - May contain internal branches and joins
  - Each dynamic wave has a wave number
    - Every value has a wave number
    - Like tagged tokens
    - Same wave number for all instructions in a wave
    - WAVE-ADVANCE instruction at top of wave
    - Wave numbers then percolate through rest of wave
- Indirect Jumps
  - For dynamic linking, procedure returns, etc.
  - INDIRECT SEND Instructions
    - Takes instruction address as one of its arguments

Memory Ordering

- Implements wave-ordered memory
- Each memory operation is annotated with ordering info
  - Wave number
  - Compiler-generated sequence no. = relative to wave
    - Traverse control flow graph breadth-first
    - Assign sequential numbers to instructions within a basic block
    - Label with predeccessor, this, successor
- Memory system reconstructs "correct" order
- May need memory no-op inst. to fill "gaps"
WaveCache

- WaveCache is the processor implementation
- Grid of approx. 2K PEs in clusters of 16 each
- Each PE has buffering and storage for 8 instructions
- Total 16K instructions (similar capacity to 64KB I-cache)
- Input queues 16 deep; indexed relative to current wave
  - Full/empty bits
  - Updated by match unit at input

Memory Hierarchy

- L1 Data Cache per 4 clusters
- L2 cache, shared conventional
- Interconnect
  - Set of shared busses w/in cluster
  - Dynamically routed network for inter-cluster comm.
- Store Buffers
  - Distributed
  - Each set of four clusters shares a store buffer
  - Each dynamic wave is bound to a store buffer
  - MEM-COORDINATE instruction acquires a store buffer for a wave and passes number to memory insts. in wave
  - OR Hash table managed in main memory

Example: Compiler Output

- Compiler steps:
  - Decompose df graph into waves
  - Insert memory seq. nos. and MEMORY-NOPs
  - Insert WAVE-ADVANCE instructions
  - Convert branch instructions into predicates (de-mux)

Example: Stable Working Set

- Encoding
  - Opcode
  - List of targets
    - Address of destn. inst.
    - Input number to which output values are directed
  - Three inputs, two outputs

- Loading, finding, placing insts.
  - Miss causes instruction to be loaded into WaveCache
  - Rewrite targets to point to locations in WaveCache
    - If target not in WaveCache, mark as “not-loaded”
    - Access to “not-loaded” instruction will cause additional misses
  - Cache miss is heavyweight
  - Must move state queue state, mark input insts. as “not-loaded”, etc.
  - "Idle" instructions are easier to replace (empty queue state)
  - Eventually instruction working set stabilizes

- Program termination
  - OS must forcibly remove instructions belonging to a process
Performance Evaluation

- Use Wavescalar system as described earlier
- Compare with absurd superscalar
  - 16-way issue, 1024 issue buffer, 1024 physical registers, etc.
  - with realistic branch predictor
  - No memory dependence speculation (in either system)
- Use ideal L1 data caches
- Separate store addresses/data (as often done in superscalar pros.)
- No speculation on memory dependences
  - Authors give results for perfect predicate prediction and memory disambiguation
  - BUT, don’t discuss how it would be implemented
- Benchmarks
  - vpr, twolf, mcf, equake, art, mpeg2, fft
  - select functions that consume 95% of execution time
  - As measured on what machine?
  - A good way to beat Amdahl’s law
- Performance results in Alpha equivalent instructions per cycle

Performance Comparison vs. Supersc.

- Avg. 3.1 faster than SS
  - over 4x on vectorizable apps (equake and fft)
  - Claim this is because wavescalar is not constrained by control dependences
  - Might get same effect in SS by using Java-like convergent path rules (research topic)
- 20-140% static instruction count overhead
- 16 PE clusters work well
  - fewer than 15% of values leave the cluster of origin
  - fewer than 10% go more than one cluster away

Performance as fcn of Cluster Size

- From 1 PE to 256 PE clusters
- 128 to 32K instructions
- Intra-cluster communication goes up for larger clusters (1 to 24 cycles)
- Performance peaks and drops off