WaveScalar: the Executive Summary

Dataflow machine
- good at exploiting ILP (dataflow parallelism)
- also traditional coarser-grain parallelism
  - cheap thread management
- low operand latency because of a hierarchical organization
- memory ordering enforced through wave-ordered memory
  - no special languages
WaveScalar

Additional motivation:

- increasing disparity between computation (fast transistors) & communication (long wires)
- increasing circuit complexity
- decreasing fabrication reliability
Monolithic von Neumann Processors

A phenomenal success today. But in 2016?

- **Performance**
  Centralized processing & control
  Long wires
  e.g., operand broadcast networks

- **Complexity**
  40-75% of “design” time is design verification

- **Defect tolerance**
  1 flaw -> paperweight
WaveScalar’s Microarchitecture

Good performance via distributed microarchitecture 😊
  • hundreds of PEs
  • organized hierarchically for fast communication between neighboring PEs
  • short point-to-point (producer to consumer) operand communication
  • dataflow execution – no centralized control
  • scalable

Low design complexity through simple, identical PEs 😊
  • design one & stamp out thousands

Defect tolerance 😊
  • route around a bad PE
Processing Element

- Simple, small (.5M transistors)
- 5-stage pipeline (receive input operands, match tags, instruction issue, execute, send output)
- Holds 64 (decoded) instructions
- 128-entry token store
- 4-entry output buffer
PEs in a Pod

- Share operand bypass network
- Back-to-back producer-consumer execution across PEs
- Relieve congestion on intra-domain bus
Domain
Cluster

D$  NS  SB
WaveScalar Processor

Long distance communication
- dynamic routing
- grid-based network
- 2-cycle hop/cluster
Whole Chip

- Can hold 32K instructions
- Normal memory hierarchy
- Traditional directory-based cache coherence
- ~400 mm$^2$ in 90 nm technology
- 1GHz.
- ~85 watts
WaveScalar Instruction Placement

operand latency vs. parallelism (resource conflicts)
WaveScalar Instruction Placement

Place instructions in PEs to maximize data locality & instruction-level parallelism.

- Instruction placement algorithm based on a performance model that captures the important performance factors
- Carve the dataflow graph of instructions into segments
  - a particular depth to make chains of dependent instructions that will be placed in the same pod
  - a particular width to make multiple independent chains that will be placed in different, but near-by pods
  - called DAWG (‘Deep and Wide Graph’ Placement)
- Snakes segments across PES in the chip on demand
- K-loop bounding to prevent instruction “explosion”
Example to Illustrate the Memory Ordering Problem

\[ A[j + i \times i] = i; \]
\[ b = A[i \times j]; \]
Example to Illustrate the Memory Ordering Problem

\[
\begin{align*}
A[j + i*i] &= i; \\
b &= A[i*j];
\end{align*}
\]
Example to Illustrate the Memory Ordering Problem

\[ A[j + i\times i] = i; \]

\[ b = A[i\times j]; \]
Wave-ordered Memory

- Compiler annotates memory operations
  - Sequence #
  - Successor
  - Predecessor
- Send memory requests in any order
- Hardware reconstructs the correct order
Wave-ordering Example

Load 2 3 4
Store 3 4 ?

Load 4 7 8
Store ? 8 9

Spring 2008  CSE 471 - WaveScalar
Wave-ordering Example

Load 2 3 4
Store 3 4 ?

4 5 6
5 6 8

Load 4 7 8
Store ? 8 9

Store buffer

2 3 4
3 4 ?

Spring 2008
CSE 471 - WaveScalar
Wave-ordering Example

Store buffer

Load 2 3 4

Store 3 4 ?

Load 4 7 8

Store ? 8 9
Wave-ordering Example

Store buffer

Load 2 3 4
Store 3 4 ?
Load 4 7 8
Store ? 8 9

4 5 6
5 6 8
Wave-ordered Memory

**Waves** are loop-free sections of the dataflow graph.

Each *dynamic* wave has a **wave number**.
Wave number is incremented between waves.

Ordering memory in a whole program:
- wave-numbers
- sequence number within a wave
WaveScalar Tag-matching

WaveScalar tag
- thread identifier
- wave number

Token: tag & value

\(<\text{ThreadID:Wave#}>\) value

\(<2:5>.3\) \(\rightarrow\) \(<2:5>.6\)

\(<2:5>.9\)
Single-thread Performance
Single-thread Performance per Area

Performance per area

AIPC/mm²

WS
OOO

ammn, art, equake, gzip, mcf, twolf, djpeg, mpeg2encode, rawaudio, average

Spring 2008
Multithreading the WaveCache

Architectural-support for WaveScalar threads

- instructions to start & stop memory orderings, i.e., threads
- memory-free synchronization to allow exclusive access to data (thread communicate instruction)
- fence instruction to force all previous memory operations to fully execute (to allow other threads to see the results of this one’s memory ops)

Combine to build threads with multiple granularities

- coarse-grain threads: 25-168X over a single thread; 2-16X over CMP, 5-11X over SMT
- fine-grain, dataflow-style threads: 18-242X over single thread
- a demonstration that one can combine the two in the same application (equake): 1.6X or 7.9X -> 9X
Creating & Terminating a Thread
Thread Creation Overhead

![Graph showing the relationship between Dependence Distance (iterations) and Loop body length (instructions) for different speedups (1x, 2x, 4x). Points A, B, and C represent specific data points.](image)
Performance of Coarse-grain Parallelism
CMP & SMT Comparison

Speedup vs 1-thread scalar CMP

- 32 threads
- 16 threads
- 8 threads
- 4 threads
- 2 threads
- 1 thread

lu, fft, radix
Relies on:
Cheap synchronization
Load once, pass data (not load/compute/store)

Performance of Fine-grain Parallelism
Building the WaveCache

RTL-level implementation
- some didn’t believe it could be built in a normal-sized chip
- some didn’t believe it could achieve a decent cycle time and load-use latencies
- Verilog & Synopsis CAD tools

Different WaveCache’s for different applications
- 1 cluster: low-cost, low power, single-thread or embedded
  - 42 mm² in 90 nm process technology, 2.2 AIPC on Splash2
- 16 clusters: multiple threads, higher performance: 378 mm², 15.8 AIPC

Board-level FPGA implementation
- OS & real application simulations