WaveScalar
Good old days
Good old days ended in Nov. 2002

- Complexity
- Clock scaling
- Area scaling
Chip MultiProcessors

- Low complexity
- Scalable
- Fast
CMP Problems

- Hard to program
- Not practical to scale
  - There only ~8 threads
- Inflexible allocation
  - Tile = allocation
- Thread parallelism only
What is WaveScalar?

- WaveScalar is a new, scalable, highly parallel processor architecture
  - Not a CMP
  - Different algorithm for executing programs
  - Different hardware organization
WaveScalar Outline

- Dataflow execution model
- Hardware design
- Evaluation
- Exploiting dataflow features
- Beyond WaveScalar: Future work
Execution Models: Von Neumann

- Von Neumann (CMP)
  - Program counter
  - Centralized
  - Sequential
Execution Model: Dataflow

- Not a new idea [Dennis, ISCA’75]
- Programs are dataflow graphs
- Instructions fire when data arrives
  - Instructions act independently
  - *All* ready instructions can fire at once
  - Massive parallelism
- Where are the dataflow machines?
Von Neumann example

\[
A[j + i*i] = i; \\
\text{Load } b \leftarrow A[i*j]; \\
\text{Mul } t1 \leftarrow i, j \\
\text{Mul } t2 \leftarrow i, i \\
\text{Add } t3 \leftarrow A, t1 \\
\text{Add } t4 \leftarrow j, t2 \\
\text{Add } t5 \leftarrow A, t4 \\
\text{Store } (t5) \leftarrow i \\
\text{Load } b \leftarrow (t3)
\]
Dataflow example

A[j + i*i] = i;
b = A[i*j];
Dataflow example

\[ A[j + i^2] = i; \]

\[ b = A[i^2j]; \]
Dataflow example

\[ A[j + i^2] = i; \]

\[ b = A[i^2j]; \]
Dataflow example

A[j + i*i] = i;

b = A[i*j];
Dataflow example

$A[j + i*i] = i;$

$b = A[i*j];$
Dataflow example

A[j + i*i] = i;

b = A[i*j];
Dataflow’s Achilles’ heel

- No ordering for memory operations
- No imperative languages (C, C++, Java)
- Designers relied on functional languages instead

To be useful, WaveScalar must solve the dataflow memory ordering problem
WaveScalar’s solution

- Order memory operations
- Just enough ordering
- Preserve parallelism
Wave-ordered memory

- Compiler annotates memory operations
  - Green: Sequence #
  - Red: Successor
  - Blue: Predecessor
- Send memory requests in any order
- Hardware reconstructs the correct order

![Diagram of wave-ordered memory operations with sequence numbers and arrows indicating loads and stores.]

- Load 2 3 4
- Store 3 4 ?
- Store 5 6 8
- Load 4 7 8
- Store ? 8 9
Wave-ordering Example
Wave-ordered Memory

- Waves are loop-free sections of the control flow graph
- Each dynamic wave has a wave number
- Each value carries its wave number
- Total ordering
  - Ordering between waves
  - “linked list” ordering within waves

[MICRO’03]
Wave-ordered Memory

- Annotations summarize the CFG
- Expressing parallelism
  - Reorder consecutive operations
- Alternative solution: token passing [Beck, JPDC’91]
  - 1/2 the parallelism
WaveScalar’s execution model

- Dataflow execution
- Von Neumann-style memory
- Coarse-grain threads
- Light-weight synchronization
WaveScalar Outline

- Execution model
- Hardware design
  - Scalable
  - Low-complexity
  - Flexible
- Evaluation
- Exploiting dataflow features
- Beyond WaveScalar: Future work
Executing WaveScalar

- Ideally
  - One ALU per instruction
  - Direct communication
- Practically
  - Fewer ALUs
  - Reuse them
WaveScalar processor architecture

- Array of processing elements (PEs)
- Dynamic instruction placement/eviction
Processing Element

- Simple, small
  - 0.5M transistors
- 5-stage pipeline
- Holds 64 instructions
PEs in a Pod
Domain
Cluster
WaveScalar Processor
WaveScalar Processor

- Long distance communication
  - Dynamic routing
  - Grid-based network
- 32K instructions
- ~400mm$^2$ 90nm
- 22FO4 (1Ghz)
WaveScalar processor architecture

- Low complexity
- Scalable
- Flexible parallelism
- Flexible allocation
Demo
Previous dataflow architectures

- Many, many previous dataflow machines
  - [Dennis, ISCA’75]
  - TTDA [Arvind, 1980]
  - Sigma-1 [Shimada, ISCA’83]
  - Manchester [Gurd, CACM’85]
  - Epsilon [Grafe, ISCA’89]
  - EM-4 [Sakai, ISCA’89]
  - Monsoon [Papadopoulos, ISCA’90]
  - *T [Nikhil, ISCA’92]
Previous dataflow architectures

- Many, many previous dataflow machines
  - [Dennis, ISCA’75]
  - TTDA [Arvind, 1980]
  - Sigma-1 [Shimada, ISCA’83]
  - Manchester [Gurd, CACM’85]
  - Epsilon [Grafe, ISCA’89]
  - EM-4 [Sakai, ISCA’89]
  - Monsoon [Papadopoulos, ISCA’90]
  - *T [Nikhil, ISCA’92]

Modern technology

WaveScalar architecture
WaveScalar Outline

- Execution model
- Hardware design
- Evaluation
  - Map WaveScalar’s design space
  - Scalability
  - CMP comparison
- Exploiting dataflow features
- Beyond WaveScalar: Future work
Performance Methodology

- Cycle-level simulator
- Workloads
  - SpecINT + SpecFP
  - Splash2
  - Mediabench
- Binary translator from Alpha -> WaveScalar
- Alpha Instructions per Cycle (AIPC)
- Synthesizable Verilog model
WaveScalar’s design space

- Many, many parameters
  - # of clusters, domains, PEs, instructions/PE, etc.
  - Very large design space
- No intuition about good designs
- How to find good designs?
  - Search by hand
  - Complete, systematic search
WaveScalar’s design space

- Constrain the design space
  - Synthesizable RTL model -> Area model
  - Fix cycle time (22FO4) and area budget (400mm²)
  - Apply some “common sense” rules
  - Focus on area-critical parameters
- There are 201 reasonable WaveScalar designs
  - Simulate them all
WaveScalar’s design space

[ISCA’06]
Pareto Optimal Designs

[ISCA'06]
WaveScalar is Scalable

7x apart in area and performance
Area efficiency

- Performance per silicon: IPC/mm²
- WaveScalar
  - 1-4 clusters: 0.07
  - 16 clusters: 0.05
- Pentium 4: 0.001-0.013
- Alpha 21264: 0.008
- Niagara (8-way CMP): 0.01
WaveScalar Outline

- Execution model
- Hardware design
- Evaluation
- Exploiting dataflow features
  - Unordered memory
  - Mix-and-match parallelism
- Beyond WaveScalar: Future work
The Unordered Memory Interface

- Wave-ordered memory is restrictive
- Circumvent it
  - Manage (lack-of) ordering explicitly
  - Load_Unordered
  - Store_Unordered
- Both interfaces co-exist happily
- Combine with fine-grain threads
  - 10s of instructions
Exploiting Unordered Memory

- Fine-grain intermingling

```c
struct {
    int x, y;
} Pair;

foo(Pair *p, int *a, int *b) {
    Pair r;
    *a = 0;
    r.x = p->x;
    r.y = p->y;
    return *b;
}
```
Exploiting Unordered Memory

- Fine-grain intermingling

```c
struct {
    int x, y;
} Pair;

foo(Pair *p, int *a, int *b) {
    Pair r;
    *a = 0;
    r.x = p->x;
    r.y = p->y;
    return *b;
}
```
Exploiting Unordered Memory

- Fine-grain intermingling

```c
struct {
    int x, y;
} Pair;

foo(Pair *p, int *a, int *b) {
    Pair r;
    *a = 0;
    r.x = p->x;
    r.y = p->y;
    return *b;
}
```
Exploiting Unordered Memory

- Fine-grain intermingling

```c
struct {
    int x, y;
} Pair;

void foo(Pair *p, int *a, int *b) {
    Pair r;
    *a = 0;
    r.x = p->x;
    r.y = p->y;
    return *b;
}
```

![Diagram showing memory access and operations]
Exploiting Unordered Memory

- Fine-grain intermingling

```c
struct {
    int x, y;
} Pair;

foo(Pair *p, int *a, int *b) {
    Pair r;
    *a = 0;
    r.x = p->x;
    r.y = p->y;
    return *b;
}
```
Exploiting Unordered Memory

- Fine-grain intermingling

```c
struct {
    int x, y;
} Pair;

foo(Pair *p, int *a, int *b) {
    Pair r;
    *a = 0;
    r.x = p->x;
    r.y = p->y;
    return *b;
}
```
Exploiting Unordered Memory

- Fine-grain intermingling

```c
struct {  
  int x, y;
} Pair;

foo(Pair *p, int *a, int *b) {
  Pair r;
  *a = 0;
  r.x = p->x;
  r.y = p->y;
  return *b;
}
```

```
Mem_nop_ack <1,2,3>
Ld p->x
Ld p->y
St r.x
St r.y
```

```
Mem_nop_ack <2,3,4>
Ld *b <3,4,5>
```
Exploiting Unordered Memory

- Fine-grain intermingling

```c
struct {
    int x, y;
} Pair;

foo(Pair *p, int *a, int *b) {
    Pair r;
    *a = 0;
    r.x = p->x;
    r.y = p->y;
    return *b;
}
```
Exploiting Unordered Memory

- Fine-grain intermingling

```c
struct {
    int x, y;
} Pair;

foo(Pair *p, int *a, int *b) {
    Pair r;
    *a = 0;
    r.x = p->x;
    r.y = p->y;
    return *b;
}
```
Putting it all together: Equake

- Finite element earthquake simulation
- >90% execution is in two functions
  - Sim()
    - Easy-to-parallelize loops
    - Coarse-grain threads
  - Smvp()
    - Cross-iteration dependences
    - Fine-grain threads + unordered memory
Putting it all together: Equake

![Graph showing speedup for different operations.]

- Single-threaded: 1
- Sim(): 1.4 (3.5)
- smvp(): 2.2 (11)
- Sim()+smvp(): 6
Conclusion

- Low complexity dataflow architecture
- Solves dataflow memory ordering
- Hybrid memory and threading interfaces
- Scalable high performance
- First ISA to encode program structure
Dataflow performance
Multithreaded Performance

The graph illustrates the speedup of various applications (fft, lu, ocean, radix, raytrace) when executed with different thread counts (1, 2, 4, 8, 16, 32, 64, 128). The x-axis represents the applications, while the y-axis shows the speedup. The average speedup across all applications is also depicted. The graph shows significant speedup improvements with increased thread counts for most applications, indicating the benefits of multithreading.
Single Thread Performance

![Performance Graph]

- ammp
- art
- equake
- gzip
- mcf
twolf
- dipeg
- mpeg2encode
rawaudio
average

WS

OOO
Single Thread Performance/mm²

Performance per area

AIPC/mm²

WS
OOO
Wave-ordered Parallelism

<table>
<thead>
<tr>
<th>Store</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Load</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Load</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Store</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>
Scaling to 4 clusters

Max performance

\[ \text{aipc/mm}^2 = 0.04 \]

Max AIPC/mm\(^2\)

\[ \text{AIPC/mm}^2 = 0.06 \]

\[ \text{Max AIPC/mm}^2 \]

\[ \text{370mm}^2 \]

\[ 8.2 \text{ AIPC} \]

\[ 0.02 \]

\[ \text{219mm}^2 \]

\[ 8.6 \text{ AIPC} \]

\[ 0.04 \]
Scaling to 16 clusters

Max AIPC/mm²

There is no scalable tile!