Scheduled Dataflow  
A Multithreaded Architecture

Krishna Kavi  
Professor and Eminent Scholar  
Department of Electrical and Computer Engineering  
The University of Alabama in Huntsville  
kavi@ece.uah.edu

http://www.eb.uah.edu/~kavi/kavi-homepage.html
What is a dataflow architecture?

Consider The Following Example

- **Dataflow**
  
  0: In 4\_L, 5\_L    -- Read X  
  1: In 4\_R, 5\_R    -- Read Y  
  2: In 6\_L        -- Read A  
  3: In 6\_R         -- Read B  
  4:  + 7\_L        -- (X+Y)  
  5:  - 7\_R        -- (X - Y)  
  6:  + 8\_R        -- (A + B)  
  7:  * 8\_L        -- (X+Y)*(X-Y)  
  8: /, Out

- **Conventional**
  
  0: Load R1, X  
  1: Load R2, Y  
  2: Load R3, A  
  3: Load R4, B  
  4:  + R1, R2, R5   (R5 = R1 + R2)  
  5:  - R1, R2, R6   (R6 = R1 - R2)  
  6:  + R3, R4, R7   (R7 = R3 + R4)  
  7:  * R5, R6, R8   (R8 = R5 * R6)  
  8: / R8, R7, R9   (R9 = R8 / R9)  
  9: Store R9

\[(X^2 - Y^2) / (A + B)\]
Features of dataflow

Data Driven ---- Instructions are enabled for execution *when and only when* operands are made available by preceding instructions
(We are changing this as explained later)

No Variables -- only Data
Results are sent directly to instructions

Freedom From Side-Effects
Functional Execution
Fine-Grained parallelism
Each instruction is an independent context

In a conventional architecture, the availability of the operands is implied by the sequencing of instructions
Early implementations of dataflow

Static Dataflow

<table>
<thead>
<tr>
<th>Operation Code</th>
<th>Avail</th>
<th>Operand</th>
<th>Avail</th>
<th>Operand</th>
<th>Destination</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>- 1</td>
<td></td>
<td>- 2</td>
<td>- 1</td>
<td>- 2</td>
</tr>
</tbody>
</table>

Instruction Format

Limitations
Only one instance of the instruction can be active.
Cannot execute loop bodies in parallel

Alternative?
Separate the operands from the instructions and associate operands with instructions and an activation number (or iteration number).

Dynamic Dataflow
Tagged Token Dataflow
Early implementations of dataflow

Tagged Token Dataflow

Each operand will be “tagged” to identify the instruction and a context (or loop iteration) to which it is destined. The tokens (data + tag) will be stored in a separate memory -- separate from instructions.

If two tokens have the same tag, they are destined to the same instruction -- a matched pair.

Matching in Tagged Token architecture used associative memory.

<table>
<thead>
<tr>
<th>PE#</th>
<th>Context (iteration #)</th>
<th>Instruction Id</th>
<th>port</th>
<th>Data Value</th>
</tr>
</thead>
</table>

SDF (Kavi)
Limitations of the dataflow model

Memory Hierarchies cannot be used
Data is sent directly to instructions as tokens

• Too fine-grained
  Each instruction is a thread

• Localities are difficult to synthesize
  Instructions may have to be fetched in “random” order

• Asynchronous execution
  An instruction is enabled when and immediately when both operands are available
Solution - 1. Introduction of memory into dataflow

Explicit Token Store Architecture (ETS)

Consider the instruction format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Offset (R)</th>
<th>Dest-1 and Port</th>
<th>Dest-2 and Port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each instruction designates a memory address where its operands will be received and “matched” -- the offset R

Results are sent to destination instructions as tokens.
Solution-2. Coarser-grained threads

ETS Code Blocks. -- A loop body or a function is treated as a code block. Can be viewed as a “coarser-grained” thread.

In actual implementations, a code-block may consist of several non-blocking threads.

<table>
<thead>
<tr>
<th>opcode</th>
<th>r</th>
<th>dests</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>2</td>
<td>+1,+2L</td>
</tr>
<tr>
<td>NEG</td>
<td>-</td>
<td>+6</td>
</tr>
<tr>
<td>SUB</td>
<td>3</td>
<td>+1</td>
</tr>
</tbody>
</table>

Instruction Memory

Frame Memory

Presence Bits

4.24
Solution 3. Synthesizing localities

Spatial Locality by grouping instructions of a code block

Temporal Locality based on multiple activations of a code block

Now we can design caches for dataflow

Data Caches present some challenges
An implementation of ETS with Caches
What are I-structures?

Used to store Arrays (or other data structures)

Single assignment is still maintained

Instructions needed:
  Allocate (A, N)
  I-Store (A, I, Value)
  I-Fetch (A, I)
A multiprocessor environment for ETS
Solution 4. Synchronous execution of dataflow

ETS Executes Instructions Asynchronously-- may need 2 cycles per binary instruction. Such architectures are called *token-driven*.

How can we execute dataflow instructions synchronously -- requiring only one cycle per instruction? (That is, make them *instruction-driven*).

1. Do not execute instructions immediately when operands are available. Hold both operands of a dataflow instruction until the instruction is scheduled.
2. Assure that when an instruction is scheduled, both operands are available.

### Operand Memory or Registers

<table>
<thead>
<tr>
<th></th>
<th>Left Port</th>
<th></th>
<th>Right Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Digression: Decoupled memory access

Separate Processor to handle all memory accesses
The earliest suggestion by J.E. Smith -- DAE architecture (1982)
Limitations of Smith’s DAE processor

- Designed for STRETCH system with no pipelines
  Single instruction stream
- Instructions for Execute processor must be coordinated with the data accesses performed by Access processor
  Very tight synchronization needed
- Coordinating conditional branches complicates the design
- Generation of coordinated instruction streams for Execute and Access may prevent traditional compiler optimizations
More recent implementations

A multithreaded processor
Separate Memory and Execution Pipelines
A thread is handed off to Memory processor when a Memory Access Instruction is decoded
A thread is handed off to Execute processor when a non-memory access instruction is decoded

Other context switches may be needed
Switch on Use -- data dependencies
Synchronization

Rhamma Processor
(Univ. Karlsruhe)
Limitations of Rhamma Processor

• Blocking Thread Model
  Requires More context switches

• Checking for data dependencies requires complex hardware

• Bubbles in pipelines are unavoidable on context switches and cache misses
More recent implementations

Pre-Load/Post-Store Processor

• A non-blocking multithreaded processor
• Separate Memory and Execution Pipelines
• A thread is enabled for execution only after all data is loaded into registers
• Storing of data is delayed until the thread completes execution
• Branch instructions cause new threads
### A simple example

<table>
<thead>
<tr>
<th></th>
<th>Conventional</th>
<th>New Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F0, 0(R1)</td>
<td>LD F0, 0(R1)</td>
</tr>
<tr>
<td>LD</td>
<td>F6, -8(R1)</td>
<td>LD F6, -8(R1)</td>
</tr>
<tr>
<td>MULTD</td>
<td>F0, F0, F2</td>
<td>LD F4, 0(R2)</td>
</tr>
<tr>
<td>MULTD</td>
<td>F6, F6, F2</td>
<td>MULTD F6, F6, F2</td>
</tr>
<tr>
<td>LD</td>
<td>F4, 0(R2)</td>
<td>MULTD F0, F0, F2</td>
</tr>
<tr>
<td>LD</td>
<td>F8, -8(R2)</td>
<td>MULTD F6, F6, F2</td>
</tr>
<tr>
<td>ADDD</td>
<td>F0, F0, F4</td>
<td>SUBI R2, R2, 16</td>
</tr>
<tr>
<td>ADDD</td>
<td>F6, F6, F8</td>
<td>ADDD F6, F6, F8</td>
</tr>
<tr>
<td>SUBI</td>
<td>R2, R2, 16</td>
<td>ADDD F0, F0, F4</td>
</tr>
<tr>
<td>SUBI</td>
<td>R1, R1, 16</td>
<td>ADDD F6, F6, F8</td>
</tr>
<tr>
<td>SD</td>
<td>8(R2), F0</td>
<td>SD 8(R2), F0</td>
</tr>
<tr>
<td>BNEZ</td>
<td>R1, LOOP</td>
<td>SD 0(R2), F6</td>
</tr>
<tr>
<td>SD</td>
<td>0(R2), F6</td>
<td></td>
</tr>
</tbody>
</table>

SDF (Kavi) 19
Features of PL/PS

- Multiple hardware contexts
- No pipeline bubbles due to cache misses
- Overlapped execution of threads
- Opportunities for better data placement and prefetching
- Fine-grained threads -- A limitation?
- Multiple hardware contexts add to hardware complexity

If 35% of instructions are memory access instructions, PL/PS can achieve 35% increase in performance with sufficient thread parallelism and completely mask memory access delays!
Back to dataflow architectures: Scheduled Dataflow

- Brings dataflow closer to conventional RISC architecture
- Utilizes Decoupled processors to eliminate pipeline bubbles on cache misses -- combines Preload/post-store with dataflow
- Eliminates WAR and WAW dependencies in pipelines
  - The result of using dataflow execution
- Uses Non-blocking Multithreaded model
Each instruction is associated with a pair of source registers. Predecessor instructions store their results in these registers.

An instruction is not enabled immediately when the two source registers are loaded. Instructions are scheduled similar to conventional processors. However, instructions retain functional properties.
Decoupled processors for Scheduled Dataflow

Execute Processor

- PC
- Context
- Instruction Fetch
- Operand Fetch
- Execute
- Write Back

Instr. Cache
Register Files

Synchronization Processor

- Preloaded Threads
- I-Strct Cache
- Operand Cache
- Synch Processor pipe
- Post Store Threads
Preliminary performance comparisons

- Monte Carlo simulations using simple models for Rhamma, Scheduled Dataflow, ETS, conventional RISC processors and other Hybrid dataflow/control-flow architectures.

- Some of the parameters are based on published data (% load/stores, avg memory latency, cache miss rates).

- Some parameters are based on simple programs coded in our architecture.

- Some parameters are based on guesswork.
Rhamma vs Scheduled Dataflow

- Multithreaded architectures (Rhamma and SDF) perform poorly for small degrees of parallelism.
- Conventional architecture is assumed to be single threaded.
- SDF is non-blocking and incurs no context switches during execution.
- SDF is dataflow based and has no WAW/WAR dependencies unlike Rhamma and Conventional architectures.

SDF (Kavi)

Effect Of Thread Level Parallelism

L is Latency and it is set to 1, 3, and 5 times the Thread run lengths.
• SDF is finer-grained. But modest thread run-lengths of 20 instruction are sufficient to outperform Rhamma

• Decoupling is the main reason for the lack of performance losses even when load/store instructions dominate
Rhamma vs Scheduled Dataflow

Effect Of Cache Misses and Miss Penalties

(a) Impact of miss rates
- SDF permits for data alignment and prefetching leading to lower cache misses
- Preload/Post store eliminates unnecessary context switches during thread execution

(b) Impact of miss penalties
ETS vs Scheduled Dataflow

- ETS requires two cycles for binary instructions while SDF architecture requires only one cycle.

- Convertional processor encounters context switches due to data dependencies and cache misses

- SDF uses a separate processor for thread scheduling

Note in ETS, each instruction is a thread but we used the same total number of instructions executed by all architectures as the normalized workload
ETS vs Scheduled Dataflow

Effect Of Thread Granularity

Except for very small threads (R) SDF outperforms ETS

For a given workload, a balance between thread granularity and thread level parallelism is needed.
ETS vs Scheduled Dataflow

Effect of Memory Access Time

- Decoupling is the main reason for the lack of significant performance loss in SDF even when load/store instructions dominate.
- ETS has no memory access instructions.
- ETS and SDF are dataflow based -- have no WAR/WAW dependencies unlike conventional architectures.
Utilization of the two processors in SDF

- Except for very small threads ($R < 10$), Synchronization Processor is not a bottleneck.

- Even with modest thread parallelism, both processors are highly utilized.
The degree of parallelism (or hardware contexts) needed to achieve high EP utilization (80%) is modest.

The degree of parallelism (or hardware contexts) needed to achieve high EP utilization depends on scheduling overhead.

SDF (Kavi)

32
Conclusions

- Combined Dataflow Architecture With Conventional control-flow like scheduling and Decoupled memory accesses
- The performance gains are primarily due to
  - Scheduling of instructions (unlike ETS)
  - Overlapped Memory/Execute processing
  - Non-Blocking and fine grained threads
  - Pre-load/Post-Store Decoupling
    - Permits for data placement and prefetching
- Eliminates Complex Instruction Scheduling hardware
  - For register renaming, detecting WAR/WAW dependencies, Branch prediction
- Fine-grained parallelism need not be expensive
- Modest number of register contexts (or thread parallelism) is sufficient
Current status and future research

- A detailed instruction simulator is being designed
- Converting Compiler backends to generate code for SDF
- Should be able to evaluate the architecture more thoroughly using large benchmarks
  Not just SPEC, but special purpose and embedded applications
- Investigate compiler optimizations
  Data placement/prefetch
  Predictive preloading
- Estimate hardware savings