INTRODUCTION ON DATAFLOW ARCHITECTURES AND TRENDS

JOSE M MONSALVE DIAZ
Postdoctoral Researcher
Mathematics and Computer science
Argonne National Laboratory

SIDDHISANKET (SID) RASKAR
Postdoctoral Researcher
Argonne Leadership Computing Facility
Argonne National Laboratory

July 31st, 2023
St. Charles, IL
OUTLINE

Introduction on Dataflow Architectures and Trends

- Von Neumann vs Dataflow
- Dataflow Model of computation
- Evolution of Dataflow architectures and Advanced Concepts
- Modern Architectures
- Challenges of Dataflow architectures
VON NEUMANN VS DATAFLOW
VON NEUMANN
SEQUENTIAL VON NEUMANN ARCHITECTURES

Instructions

\[ a = 1 \]
\[ b = 3 \]
\[ c = 4 \]
\[ d = 3 \]
\[ r1 = a + b \]
\[ r2 = c + d \]
\[ r3 = r1 \times r2 \]
VON NEUMANN

SEQUENTIAL VON NEUMANN ARCHITECTURES

Instructions

\[ a = 1 \]
\[ b = 3 \]
\[ c = 4 \]
\[ d = 3 \]
\[ r_1 = a + b \]
\[ r_2 = c + d \]
\[ r_3 = r_1 \times r_2 \]
VON NEUMANN

SEQUENTIAL VON NEUMANN ARCHITECTURES

Instructions

\[
\begin{align*}
  a &= 1 \\
  b &= 3 \\
  c &= 4 \\
  d &= 3 \\
  r_1 &= a + b \\
  r_2 &= c + d \\
  r_3 &= r_1 \times r_2
\end{align*}
\]
VON NEUMANN
SEQUENTIAL VON NEUMANN ARCHITECTURES

Instructions

\[ a = 1 \]
\[ b = 3 \]
\[ c = 4 \]
\[ d = 3 \]
\[ r1 = a + b \]
\[ r2 = c + d \]
\[ r3 = r1 \times r2 \]
VON NEUMANN
SEQUENTIAL VON NEUMANN ARCHITECTURES

Instructions

\[ a = 1 \]
\[ b = 3 \]
\[ c = 4 \]
\[ d = 3 \]
\[ r_1 = a + b \]
\[ r_2 = c + d \]
\[ r_3 = r_1 \times r_2 \]
VON NEUMANN

SEQUENTIAL VON NEUMANN ARCHITECTURES

Instructions

\[ a = 1 \]
\[ b = 3 \]
\[ c = 4 \]
\[ d = 3 \]
\[ r1 = a + b \]
\[ r2 = c + d \]
\[ r3 = r1 \times r2 \]
VON NEUMANN
SEQUENTIAL VON NEUMANN ARCHITECTURES

Instructions

\[ a = 1 \]
\[ b = 3 \]
\[ c = 4 \]
\[ d = 3 \]
\[ r_1 = a + b \]
\[ r_2 = c + d \]
\[ r_3 = r_1 \times r_2 \]
VON NEUMANN
SEQUENTIAL VON NEUMANN ARCHITECTURES

Instructions

\[ a = 1 \]
\[ b = 3 \]
\[ c = 4 \]
\[ d = 3 \]
\[ r1 = a + b \]
\[ r2 = c + d \]
\[ r3 = r1 \times r2 \]
VON NEUMANN VS DATAFLOW

SEQUENTIAL VON NEUMANN ARCHITECTURES

Instructions

a = 1
b = 3
c = 4
d = 3
r1 = a + b
r2 = c + d
r3 = r1 * r2
DATAFLOW PROGRAMS

A program is represented as a graph. Nodes are operations. Arcs are operands that contain tokens.
DATAFLOW MODEL OF COMPUTATION

\[ a + b + c + d + e \]

\[ \begin{array}{cccc}
1 \\
3 \\
4 \\
3 \\
\end{array} \]

\[ \begin{array}{cc}
+ \\
* \\
+ \\
\end{array} \]
DATAFLOW MODEL OF COMPUTATION

\[ a + b + c + d + e \]

\[ 4 + 4 = 8 \]
\[ (4 + 3) \times 4 = 20 \]
DATAFLOW MODEL OF COMPUTATION

\[ a + b + c 	imes d + e = 7 + 4 \]
DATAFLOW MODEL OF COMPUTATION
DATAFLOW MODEL OF COMPUTATION
DATAFLOW MODEL OF COMPUTATION

- Define what a program is

- What are the operands

- What is well defined dataflow graphs?
DATAFLOW OPERATIONAL SEMANTICS

- **Tokens ➔ Data values**

- **Firing Rules ➔ All tokens are present in the input arcs**
  - Actor removes tokens from each of its input arcs
  - Actor executes operation
  - Actor places tokens on each of its output arcs

- **Assignment operation ➔ Placing a token in the output arc**
\[
x = a + b;
\]
\[
y = b \times 7;
\]
\[
z = (x-y) \times (x+y);
\]
DATAFLOW GRAPHS

\[ x = a + b; \]
\[ y = b \times 7; \]
\[ z = (x - y) \times (x + y); \]

Values in dataflow graphs represented as tokens

\(<\text{instruction}_\text{ptr}, \text{port}, \text{value}>\)
DATAFLOW GRAPHS

\[ x = a + b; \]
\[ y = b \times 7; \]
\[ z = (x - y) \times (x + y); \]

An operator executes when all its input tokens are present.
DATAFLOW GRAPHS

\[
x = a + b;
\]
\[
y = b \times 7;
\]
\[
z = (x - y) \times (x + y);
\]

No separate control flow

Copies of the result token are distributed to the destination operators
DATAFLOW OPERATIONAL SEMANTICS

- Snapshot/Configuration ➔ State of the program
- Next state ➔ Any enabled actor fired defines the “next state” of the computation

![Diagram showing the transition from snapshot to next state]

Snapshot ➔ Next state ➔ Snapshot
DATAFLOW OPERATORS

DATAFLOW OPERATORS

Control Flow operators

if ( p(y) ) {
  f(x, y);
} else {
  g(y);
}
if ( p(y) ) {
    f(x, y);
} else {
    g(y);
}
if ( p(y) ) {
  f(x, y);
} else {
  g(y);
}
if( p(y) ) {
    f(x, y);
} else {
    g(y);
}
CONDITIONAL EXPRESSIONS

```c
if ( p(y) ) {
    f(x, y);
} else {
    g(y);
}
```
if ( p(y) ) {
  f(x, y);
} else {
  g(y);
}
if( p(y) ) {
  f(x, y);
} else {
  g(y);
}
LOOP SCHEMA

for (init; condition ; increment)
{ //....// }
for (int i = 0; i < N; i ++)
{
    //....//
}
for (int i = 0; i < 2; i++)
{
    //....//
}
for (int i = 0; i < 2; i++)
{
    //....//
}
for (int i = 0; i < 2; i++)
{
    //....//
}
for (int i = 0; i < 2; i ++)
{
  //....//
}
for (int i = 0; i < 2; i++)
{
    //....//
}
for (int i = 0; i < 2; i++)
{
    //....//
}
for (int i = 0; i < 2; i++)
{
//....//
}
for (int i = 0; i < 2; i++)
{ //....// }

Initial Loop value

+1

< 2
for (int i = 0; i < 2; i++)
{
    //....//
}
for (int i = 0; i < 2; i ++)
{
    //....//
}
for (int i = 0; i < 2; i ++)
{
    //....//
}
LOOP SCHEMA

for (int i = 0; i < 2; i++)
{
    //....//
}
for (int i = 0; i < 2; i++)
{
    //....//
}
for (int i = 0; i < 2; i++) {
    //....//
}
WELL BEHAVED DATAFLOW GRAPHS

Data flow graphs that produce exactly one set of result values at each output arcs for each set of values presented at the input arcs
WELL BEHAVED DATAFLOW GRAPHS

An \((m, n)\) Schema with no enabled actors

(a) Initial Snapshot

(b) Final Snapshot
WELL BEHAVED DATAFLOW GRAPHS

Implies Initial configuration is re-established
LOOP SCHEMA IS WELL BEHAVED

Initial Configuration

Final Configuration
"SICK" DATAFLOW GRAPHS

Arbitrary connections of data flow operators can result in pathological programs

Deadlock
Hang up
Conflict
Unclean
DATAFLOW MODEL OF COMPUTATION

- Define what a program is
- What are the operands
- What is well defined dataflow graphs?
EVOLUTION OF DATAFLOW ARCHITECTURES AND CONCEPTS
Arvind & Nikhil proposed the Tagged Token Dataflow Architecture.
DATAFLOW ARCHITECTURES

Machines from the "first spring" of dataflow

Static Dataflow
Dennis 1972
MIT

MIT TTDA
Arvind 1980

LAU
Syre 1976

Manchester
Gurd & Watson 1982

Dataflow Multiprocessor
Rambaugh 1976

Arg-Fetching Dataflow
Dennis Gao 1987-88

Monsoon
Papadopoulos & Culler 1988

Iannuci’s 1988-92

P-RISC
Nikhil & Arvind 1989

TAM
Culler 1990

SIGMA-I
Shimada 1988

EM-5/4/X
RWC-1 1992-97

MDFA
Gao 1989-93

MTA
HumTheobald Gao 94

EARTH
PACT95', ISCA96, Theobald99

*T/Start-NG
MIT/Motorola 1991-
A DATAFLOW MULTIPROCESSOR
JAMES RUMBAUGH

A Data Flow Multiprocessor

JAMES RUMBAUGH

Abstract—This paper presents the architecture of a highly concurrent multiprocessor which runs programs expressed in data flow notation. Sequencing of data flow instruction execution depends only on the availability of operands required by instructions. Because data flow instructions have no side effects, unrelated instructions can be executed concurrently without interference if each has its required operands.

The data flow multiprocessor is hierarchically constructed as a network of simple modules. All module interactions are asynchronous. The principal working elements of the machine are a set of activation processors, each of which performs the execution of one invocation of a data flow procedure held in a local memory within the processor. A pipeline of logical units within each processor executes several concurrently active instructions. All data flow operations are performed within single processors except procedure calls, which cause the creation of new activations in other processors, and operations on large data structures, which are performed by structure controller modules using values stored in a central memory. Concurrency within a data flow procedure provides a processor with something to do while a slow operation is being processed.

The behavior of the machine has been specified by a formal description language and has been shown to correctly implement the data flow language. The principal advantages of the data flow multiprocessor over conventional designs are reduced complexity of the processor-memory connection, greater use of pipelining, and a simpler representation and implementation of concurrent activity.

Index Terms—Asynchronous logic, cache, concurrency, data-driven instruction execution, data flow program, modularity, multiprocessor, parallel processor, pipelining.

I. INTRODUCTION

The design of a highly concurrent computer is complicated by potential interdependencies within programs and between modules of the machine. This paper presents the architecture of a multiprocessor and an associated program notation which reduce such interdependencies. Because interactions are restricted, modules
Fig. 14. Activation processor.
RAMBAUGH’S ARCHITECTURE LOCAL MEMORY

Fig. 14. Activation processor.
Fig. 14. Activation processor.
Fig. 14. Activation processor.
RAMBAUGH’S ARCHITECTURE ACTIVATION PROCESSOR

Fig. 14. Activation processor.

Circular execution pipeline
RAMBAUGH’S ARCHITECTURE LOCAL MEMORY

### Instruction Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Op1 Addr</th>
<th>Op2 Addr</th>
<th>Res Addr</th>
<th>Succ Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>+</td>
<td>0x44</td>
<td>0x48</td>
<td>0x4C</td>
<td>0x28</td>
</tr>
<tr>
<td>0x24</td>
<td>-</td>
<td>0x50</td>
<td>0x54</td>
<td>0x58</td>
<td>0x28</td>
</tr>
<tr>
<td>0x28</td>
<td>x</td>
<td>0x4C</td>
<td>0x58</td>
<td>0x5C</td>
<td>...</td>
</tr>
</tbody>
</table>

### Enabling Count Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>0</td>
</tr>
<tr>
<td>0x24</td>
<td>0</td>
</tr>
<tr>
<td>0x28</td>
<td>2</td>
</tr>
</tbody>
</table>

### Data Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x44</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>0x48</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>0x4C</td>
<td>int</td>
<td>xxx</td>
</tr>
<tr>
<td>0x50</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>0x54</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>0x58</td>
<td>int</td>
<td>xxx</td>
</tr>
<tr>
<td>0x5C</td>
<td>double</td>
<td>xxx</td>
</tr>
</tbody>
</table>
## RAMBAUGH’S ARCHITECTURE LOCAL MEMORY

### Instruction Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Op1 Addr</th>
<th>Op2 Addr</th>
<th>Res Addr</th>
<th>Succ Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>+</td>
<td>0x44</td>
<td>0x48</td>
<td>0x4C</td>
<td>0x28</td>
</tr>
<tr>
<td>0x24</td>
<td>-</td>
<td>0x50</td>
<td>0x54</td>
<td>0x58</td>
<td>0x28</td>
</tr>
<tr>
<td>0x28</td>
<td>x</td>
<td>0x4C</td>
<td>0x58</td>
<td>0x5C</td>
<td>...</td>
</tr>
</tbody>
</table>

### Enabling Count Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>2</td>
</tr>
<tr>
<td>0x24</td>
<td>2</td>
</tr>
<tr>
<td>0x28</td>
<td>0</td>
</tr>
</tbody>
</table>

### Data Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x44</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>0x48</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>0x4C</td>
<td>int</td>
<td>7</td>
</tr>
<tr>
<td>0x50</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>0x54</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>0x58</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>0x5C</td>
<td>double</td>
<td>xxx</td>
</tr>
</tbody>
</table>
### RAMBAUGH’S ARCHITECTURE LOCAL MEMORY

#### Enabling Count Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>2</td>
</tr>
<tr>
<td>0x24</td>
<td>2</td>
</tr>
<tr>
<td>0x28</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Instruction Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Op1 Addr</th>
<th>Op2 Addr</th>
<th>Res Addr</th>
<th>Succ Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>+</td>
<td>0x44</td>
<td>0x48</td>
<td>0x4C</td>
<td>0x28</td>
</tr>
<tr>
<td>0x24</td>
<td>-</td>
<td>0x50</td>
<td>0x54</td>
<td>0x58</td>
<td>0x28</td>
</tr>
<tr>
<td>0x28</td>
<td>x</td>
<td>0x4C</td>
<td>0x58</td>
<td>0x5C</td>
<td>...</td>
</tr>
</tbody>
</table>

#### Data Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x44</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>0x48</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>0x4C</td>
<td>int</td>
<td>7</td>
</tr>
<tr>
<td>0x50</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>0x54</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>0x58</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>0x5C</td>
<td>double</td>
<td>28</td>
</tr>
</tbody>
</table>
DATAFLOW ARCHITECTURES

- Execute operational semantics of dataflow in hardware
- Represent tokens and their matching to instructions
- Memory is split into name/value pairs explicitly associated with the instructions
DATAFLOW PERFORMANCE
DATAFLOW PERFORMANCE

Critical path

longest path from $S$ to $T$ is known as the critical path.

Dataflow Minimum execution time is determined by the critical path and scheduling of ready tokens.

DATAFLOW PERFORMANCE

Critical Path

\[ f() \rightarrow f() \rightarrow f() \rightarrow f() \rightarrow f() \]

\( T = 0 \)
DATAFLOW PERFORMANCE

Critical Path

T = 1
DATAFLOW PERFORMANCE

Critical Path

\[ T = 2 \]
DATAFLOW PERFORMANCE

Critical Path

\[ T = 3 \]
DATAFLOW PERFORMANCE

Critical Path

\[ T = 4 \]
DATAFLOW PERFORMANCE

Critical Path

\[ T = 5 \]
DATAFLOW PERFORMANCE

Critical Path

\[ T = 6 \]
DATAFLOW PERFORMANCE

Critical Path

Execution time determined by the critical path

T = 6
DATAFLOW PIPELINING
DATAFLOW PIPELINING
Allow multiple executions of the same dataflow graph

\[ x^2 \rightarrow + \rightarrow 0.5x \rightarrow x \]

\[-1 \rightarrow + \rightarrow 0.5x \rightarrow x \]

\[ x^n + 0.5x - 1 \]
DATAFLOW PIPELINING
Allow multiple executions of the same dataflow graph
DATAFLOW PIPELINING
Allow multiple executions of the same dataflow graph

\[ x^2 - 1 + 0.5x \]
DATAFLOW PIPELINING
Allow multiple executions of the same dataflow graph

\[
\chi^2 - 1 + 0.5x
\]
DATAFLOW PIPELINING
Allow multiple executions of the same dataflow graph

\[ x^2 - 1 + 0.5x \]
DATAFLOW PIPELINING

Allow multiple executions of the same dataflow graph

\[ x^2 - 1 + 0.5x. \]
DATAFLOW PIPELINING

The back-pressure problem

Data Token

Data Arc

Arc must be empty
DATAFLOW PIPELINING: THE BACK-PRESSURE PROBLEM
DATAFLOW PIPELINING

The back-pressure problem

\[ x^2 - 1 + 0.5x \]

- **Backpressure Token**
- **Data Token**
- **Data Arc**
- **Back pressure**
DATAFLOW PIPELINING

The back-pressure problem

\[ x^2 - 1 + 0.5x \]

Backpressure Token
Data Token
Data Arc
Back pressure
DATAFLOW PIPELINING

The back-pressure token

\[ x^2 - 1 + 0.5x \]

Initial State
DATAFLOW PIPELINING
The back-pressure token

\[ x^2 - 1 + 0.5x - 1 \]
DATAFLOW PIPELINING

The back-pressure token

Actors can be fired since ACK tokens are present in their outputs
DATAFLOW PIPELINING

The back-pressure token

\[ x^2 - 1 + 0.5x \]

Backpressure tokens are consumed as well. The data paths are full.
DATAFLOW PIPELINING

The back-pressure token

Backpressure tokens are produced in the direction against data flowing.
DATAFLOW PIPELINING

The back-pressure token

$x^2 - 1 + 0.5x$
DATAFLOW PIPELINING

The back-pressure token

\[ x^2 - 1 + 0.5x. \]
DATAFLOW PIPELINING

The back-pressure token

Actors must wait for backpressure tokens to be fired
DATAFLOW PIPELINING

The back-pressure token
DATAFLOW PERFORMANCE:
THE UNBALANCE PROBLEM
DATAFLOW PERFORMANCE

Path unbalance

\[ f() \rightarrow f() \rightarrow f() \rightarrow f() \rightarrow f() \rightarrow f() \rightarrow f() \]

\[ T = 0 \]
DATAFLOW PERFORMANCE

Path unbalance

\[ T = 1 \]
DATAFLOW PERFORMANCE

Path unbalance

This token cannot be consumed

Because this arc is full

\[ T = 2 \]
DATAFLOW PERFORMANCE

Path unbalance

\[ T = 3 \]
DATAFLOW PERFORMANCE

Path unbalance

T = 4
DATAFLOW PERFORMANCE

Path unbalance

\[ T = 5 \]
DATAFLOW PERFORMANCE

Path unbalance

Now we can start executing the next token

T = 6
DATAFLOW PERFORMANCE: BALANCING
DATAFLOW PERFORMANCE

Queue insertion

\[ f(\cdot) \]

\[ T = 0 \]
DATAFLOW PERFORMANCE

Queue insertion

\[ T = 1 \]
DATAFLOW PERFORMANCE
Queue insertion

$T = 2$
DATAFLOW PERFORMANCE

Queue insertion

\[ T = 3 \]
DATAFLOW PERFORMANCE

Queue insertion

\[ f() \]

\[ T = 4 \]
DATAFLOW PERFORMANCE

Queue insertion

T = 5

\( f() \)

\( f() \)

\( f() \)

\( f() \)

\( f() \)
DATAFLOW PERFORMANCE

Queue insertion

\[ f() \quad f() \quad f() \quad f() \quad f() \quad f() \]

\[ T = 6 \]
DATAFLOW PERFORMANCE

Queue insertion

$T = 7$
DATAFLOW PERFORMANCE

Queue insertion

\[ f() \]

\[ T = 8 \]
DATAFLOW PERFORMANCE

Queue insertion

\[ f() \rightarrow f() \rightarrow f() \rightarrow f() \rightarrow f() \rightarrow f() \rightarrow f() \rightarrow f() \]

\[ T = 9 \]
DATAFLOW PERFORMANCE
Queue insertion

\[ f(\cdot) \rightarrow f(\cdot) \rightarrow f(\cdot) \rightarrow f(\cdot) \rightarrow f(\cdot) \]

\[ T = 10 \]
DATAFLOW PERFORMANCE
Queue insertion – Load balancing

The queue allows balancing the length of the paths

STATIC VS DYNAMIC DATAFLOW
THE RE-ENTRY PROBLEM
long fact(n) {
    if (n == 0)
        return 1;
    else
        return n * fact(n-1);
}
long fact(n) {
    if (n == 0)
        return 1;
    else
        return n * fact(n-1);
}
long fact(n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
long fact(n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}
long fact(n) {
    if(n == 0) {
        return 1;
    }
    else {
        return n * fact(n-1);
    }
}
long fact(n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}
```java
long fact(n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}
```
The function `fact(2)` is defined as follows:

```c
long fact(n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}
```

This function calculates the factorial of a number. For `fact(2)`, we have:

1. Check if `n` equals 0. If so, return 1.
2. If not, return `n` times the factorial of `n-1`.
Fact(2)

long fact(n) {
    if(n == 0)
        return 1;
    else
        return n * fact(n-1);
}
FACTORIAL

```
long fact(n) {
    if(n == 0)
        return 1;
    else
        return n * fact(n-1);
}
```

`fact(2)`
factorial(n)

```
Apply fact

n==0

F

1

T

-1

* 6

n==0

Reenter the same graph
```
FACTORIAL REENTRY PROBLEM

$fact(2)$

$fact(1)$
FACTORIAL REENTRY PROBLEM

**fact(2)**

Apply fact

**fact(1)**

Apply fact

Where did it go
DYNAMIC DATAFLOW
STATIC DATAFLOW GRAPHS

Values in static dataflow graphs represented as tokens

<instruction_ptr, port, value>
DYNAMIC DATAFLOW GRAPHS
a.k.a. Color token or tagged token dataflow

Values in **dynamic** dataflow graphs represented as tokens

<color, instruction_ptr, port, value>
There may be multiple tokens per arc, as long as they are of different “color”

\(<\text{yellow, 3, Left, 99}>\)

\(<\text{turquoise, 3, Left, 99}>\)
DYNAMIC DATAFLOW GRAPHS
a.k.a. Color token or tagged token dataflow

Operational semantics also change:

• Firing Rules ➔ All tokens of the same color are present in the input arcs.
DYNAMIC DATAFLOW
THE APPLY OPERATOR
DYNAMIC DATAFLOW

THE APPLY OPERATOR

\[ \text{Apply} \rightarrow A \rightarrow A^{-1} \rightarrow \text{END} \]

\[ \text{BEGIN} \rightarrow \text{Dataflow graph of Procedure } q \rightarrow \text{END} \]
DYNAMIC DATAFLOW
THE APPLY OPERATOR

Dataflow graph of Procedure q
DYNAMIC DATAFLOW
THE APPLY OPERATOR
DYNAMIC DATAFLOW
THE APPLY OPERATOR

\[ \text{BEGIN} \rightarrow \text{Apply} \rightarrow A \rightarrow A^{-1} \rightarrow \text{END} \]

\[ q \rightarrow a \]

Dataflow graph of Procedure q
DYNAMIC DATAFLOW
THE APPLY OPERATOR

\[ \text{Apply} \quad \equiv \quad \text{BEGIN} \]

\[ \begin{array}{ccc}
q & \rightarrow & a \\
\downarrow & & \downarrow \\
\text{Apply} & \rightarrow & \quad \\
\downarrow & & \downarrow \\
A^{-1} & \rightarrow & \quad \\
\downarrow & & \downarrow \\
\text{END} & \rightarrow & \quad \\
\downarrow & & \downarrow \\
\text{Dataflow graph of Procedure q} & \rightarrow & \quad \\
\end{array} \]
FACTORIAL REENTRY PROBLEM

fact(2)

Color change

fact(1)
Executing a Program on the MIT Tagged-Token Dataflow Architecture

AVINOD, ARVIND V., ANDamp; REINSBERGER, W., ANDamp; REINSBERGER, W.

I. Introduction

There are several commercial and research efforts currently underway to build general-purpose computers with performance beyond what is possible today. Among these approaches that can be classified as general-purpose, “multiply-nation multiple task” (MNMT) machines are evolutionary in nature. For architectures that employ interconnections of conventional von Neumann machines for programming, they rely upon conventional sequential languages (such as Fortran, C, or LISP) compiled with some degree of parallelism. These architectures are necessary because the available collection of adequate parallelism resides in a difficult problem, in spite of recent advances in compiler technology [25, 27, 28, 30].

Unfortunately, a traditional von Neumann processor has hardware characteristics that reduce its effectiveness in a parallel machine. First, its performance suffers in the presence of long memory and communication latencies, and there are unavailable in a parallel machine. Second, they do not provide good synchronization mechanisms for frequent task switching between parallel activities, again resulting in a parallel machine. The detailed technical examination of these issues may be found in [31]. In [27], the authors exhibit architectural changes to remedy these problems, inspired by dataflow architectures.

Performing traditional programming languages are not easily adapted to incorporate parallelism. First, the lack of support for dataflow graphs prohibits the expression of many concurrent tasks. Second, dataflow graphs are composed of nodes that are sequential in nature, whereas in real-time systems, there is a requirement for concurrency. Finally, it is significant and application-specific for the programmer to manage parallelism explicitly. To identify and schedule parallel tasks, even though to make the machine more effective, but large enough to keep the resource management eventually feasible.

In contrast, our dataflow approach is quite revolutionary. We begin with [27], a high-level language with fine-grained parallelism explicit in its operational semantics. Despite this potential for enormous parallelism, the semantics are also deterministic. Program in [27] are compiled into dataflow graphs, which contain a parallel machine language. Finally, dataflow graphs are executed directly in the Tagged-Token Dataflow Architecture (TTDA), a machine with highly data-driven instruction scheduling, unlike the sequential program execution

Dataflow research has moved greatly since the seminal paper on dataflow graphs by Dallmeier [18]. Many refinements have been the U-Transformer for Dynamic Dataflow graphs [32], the first version of [32], the Machine Dataflow architecture [32] and, more recently, the Tellis Parallel [48] [20]. But much has happened since then in all levels—language, compiler, and architecture—and dataflow, not being a mainstream approach, requires some innovation. In this paper, we provide an accurate snapshot as of early 1995, by providing a highly detailed explanation of the compilation and execution of an IL program. Because of the exposure to topics, our coverage of either the language and compiler or the machine architecture will address in detail the various points of interest from the interested reader.

In Section II, we present an example program expressed in IL, our high-level parallel language. We take the opportunity to explain its parallelism, and our reliance on parallel languages in general. In Section III, we explore dataflow graphs as a parallel machine language and show how to compile the example program. In Section IV, we describe the MIT Tagged-Token Dataflow Architecture and show how to execute and run dataflow graphs. Finally, in Section V,
MODERN DATAFLOW ARCHITECTURES
Arvind & Nikhil proposed the Tagged Token Dataflow Architecture

BRIEF HISTORY OF DATAFLOW

1960
Carl Adam Petri defines Petri Nets

1970
Estrin and Turn proposed an early dataflow model

1980
Rodriguez proposes Dataflow Graphs

1990
Chamberlain proposes Single Assignment language for dataflow

Karp and Miller analyzed Computation Graphs w/o branches or merges

Dennis proposes a dataflow language. Pure Dataflow is born

Kahn proposes a simple parallel processing language with vertices as queues. Static Dataflow is born

1980
Arvind, Nikhil designed the Monsoon dataflow machine

Arvind and Gostelow, & separately Gurd and Watson created a tagged token dataflow model. Dynamic Dataflow is born

Rambaugh proposes a dataflow multiprocessor architecture

2010
Arvind & Nikhil proposed the Tagged Token Dataflow Architecture

WHAT IS NEW?
1. Granularity of dataflow operations
2. Spatial reconfigurable architectures
SPATIAL RECONFIGURABLE ARCHITECTURES
COARSE GRAIN DATAFLOW

Dataflow graph node size

Dataflow graph node executes more complex mathematical operations

- Hybrid execution models
  - Von Neumann + Dataflow
- Node \( g() \) is defined in terms of multiple instructions
- Data locations (i.e., tokens) \( a, b, \) and \( c \) have a larger memory footprint

\[ c = g(a, b) \]
SOME EXAMPLES OF CURRENT DATAFLOW ARCHITECTURES
# Overview of Modern Spatial Architectures

<table>
<thead>
<tr>
<th></th>
<th>Cerebras CS2</th>
<th>SambaNova Cardinal SN30</th>
<th>Groq GroqRack</th>
<th>Habana Gaudi1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compute Units</strong></td>
<td>850,000 Cores</td>
<td>640 PCUs</td>
<td>5120 vector ALUs</td>
<td>8 TPC + GEMM engine</td>
</tr>
<tr>
<td><strong>On-Chip Memory</strong></td>
<td>40 GB L1, 1TB+ MemoryX</td>
<td>&gt;300MB L1 1TB</td>
<td>230MB L1</td>
<td>24 MB L1 32GB</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>7nm</td>
<td>7nm</td>
<td>7 nm</td>
<td>7nm</td>
</tr>
<tr>
<td><strong>System Size</strong></td>
<td>2 Nodes including Memory-X and Swarm-X</td>
<td>8 nodes (8 cards per node)</td>
<td>9 nodes (8 cards per node)</td>
<td>2 nodes (8 cards per node)</td>
</tr>
<tr>
<td><strong>Estimated Performance of a card (TFlops)</strong></td>
<td>&gt;5780 (FP16)</td>
<td>&gt;660 (BF16)</td>
<td>&gt;250 (FP16) &gt;1000 (INT8)</td>
<td>&gt;150 (FP16)</td>
</tr>
<tr>
<td><strong>Software Stack Support</strong></td>
<td>Tensorflow, Pytorch, CSLang</td>
<td>SambaFlow, Pytorch, C++ SDK</td>
<td>GroqAPI, ONNX, C/C++ Groq Runtime</td>
<td>Synapse AI, TensorFlow and PyTorch, TPC</td>
</tr>
<tr>
<td><strong>Interconnect</strong></td>
<td>Ethernet-based</td>
<td>Ethernet-based</td>
<td>RealScale™</td>
<td>Ethernet-based</td>
</tr>
</tbody>
</table>

https://www.alcf.anl.gov/alcf-ai-testbed
The old way: kernels are bottlenecked by memory bandwidth and host overhead.

The Dataflow way: Spatial dataflow eliminates memory traffic and overhead.

Simple Convolution Graph

Dataflow: Kernels are spatially mapped onto the accelerator and data flows on-chip between them reducing memory traffic.

GPU accelerators: Each kernel is launched onto the device and bottlenecks include memory bandwidth and kernel-launch latencies.

Image Courtesy: Sumti Jairath, SambaNova
HANDS ON WITH AI ACCELERATORS

Track 8 – Machine Learning
*Introductions to AI Testbed at ALCF and Hands-on*
3.30-5.00 PM, August 11
Siddhisanket (Sid) Raskar

Cerebras
Cerebras CS-2 Wafer-Scale Cluster WSE-2

SambaNova
SambaNova DataScale SN30

Graphcore
Graphcore Bow Pod64
DATAFLOW CHALLENGES
SPATIAL DATAFLOW SCHEDULING VS ARGUMENT FETCHING

NP-Hard problem (Factorial Complexity)

Runtime scheduling decision overhead
RECOMMENDATION

Two different points of view

Two Fundamental Issues in Multiprocessing

Arvind
Robert A. Iannucci
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139 - USA

Abstract

A general purpose multiprocessor should be scalable, i.e. show higher performance when more hardware resources are added to the machine. Architects of such multiprocessors must address the loss in processor efficiency due to two fundamental issues: long memory latencies and waits due to synchronization events. It is argued that a well designed processor can overcome these losses provided there is sufficient parallelism in the program being executed. The detrimental effect of long latency can be reduced by instruction pipelining, however, the restriction of a single thread of computation in von Neumann processors severely limits their ability to have more than a few instructions in the pipeline. Furthermore, techniques to reduce the memory latency tend to increase the cost of task switching. The cost of synchronization events in von Neumann machines makes decomposing a program into very small tasks counter-productive. Dataflow machines, on the other hand, treat each instruction as a task, and by paying a small synchronization cost for each instruction executed, offer the ultimate flexibility in scheduling instructions to reduce processor idle time.

Key words and phrases: caches, cache coherence, dataflow architectures, hazard resolution, instruction pipelining, LOAD/STORE architectures, memory latency, multiprocessors, multi-thread architectures, semaphores, synchronization, von Neumann architecture.

Two Fundamental Limits on Dataflow Multiprocessing

David E. Culler
Klaus Erik Schunck
Thorsten von Eicken
Report No. UCB/CSD 92/716
Computer Science Division
University of California, Berkeley

Abstract: This paper examines the argument for dataflow architectures in “Two Fundamental Issues in Multiprocessing[6].” We observe two key problems. First, the justification of extensive multithreading is based on an overly simplistic view of the storage hierarchy. Second, the local greedy scheduling policy embodied in dataflow is inadequate in many circumstances. A more realistic model of the storage hierarchy imposes significant constraints on the scheduling of computation and requires a degree of parsimony in the scheduling policy. In particular, it is important to establish a scheduling hierarchy that reflects the underlying storage hierarchy. However, even with this improvement, simple local scheduling policies are unlikely to be adequate.

Keywords: dataflow, multiprocessing, multithreading, latency tolerance, storage hierarchy, scheduling hierarchy.

1 Introduction

The advantages of dataflow architectures were argued persuasively in a seminal 1983 paper by Arvind and Iannucci[4] and in a 1987 revision entitled “Two Fundamental Issues in Multiprocessing[5].” However, reality has proved less favorable to this approach than their arguments would suggest. This motivates us to examine the line of reasoning that has driven dataflow architectures and two-stage multithreading to understand where the argument went awry. We observe two key problems. First, the justification of extensive multithreading is based on an overly simplistic view of the storage hierarchy. Second, the local greedy scheduling policy embodied in dataflow, “execute whenever data is available” is inadequate in many circumstances. A more realistic model of the storage hierarchy imposes significant constraints on the scheduling of computation and requires a degree of parsimony in the scheduling policy. In particular, it is important to establish a scheduling hierarchy that reflects the underlying storage hierarchy. However, even with this improvement, simple local scheduling policies are unlikely to be adequate.
Limit 1: Storage Hierarchy

“Dataflow architectures essentially replace the small register number with a large tag that serves to “name” the value. A realistic view of the storage hierarchy requires that only a small number of such name/value pairs can be resident at a time. Once the number of VPs exceeds the capacity of the top level matching store, the synchronization cost increases dramatically, since some form of overflow store must be used.”
TWO FUNDAMENTAL LIMITS ON DATAFLOW MULTIPROCESSING

David E. Culler et. al.

• Limit 2: Local Dynamic Scheduling

“…any naïve local scheduling policy exhibits unnecessarily low machine efficiency or high resource requirements on some programs. Thus, it would seem unwise to rely solely on low-level hardware mechanisms or runtime system support to determine the scheduling of computation.”
CHALLENGES

• Memory management
• Resource allocation and scheduling
• Parallelism control vs locality
• Balancing “size” of each dataflow node
• Programmability
• …
THANK YOU

IN LOVING MEMORY OF

Professor
Guang R Gao
1945-2021

Let his legacy be remembered and his impact persist forever