

# Scalability of Scheduled Dataflow Architecture (SDF) With Register Contexts

by  
**Joseph M. Arul**  
and  
**Krishna M. Kavi**

The 5<sup>th</sup> International Conference on Algorithms and  
Architectures for Parallel Processing  
Beijing, Oct 23<sup>rd</sup> - 25<sup>th</sup>, 2002.

# Introduction

- Control flow vs. Dataflow models of architectures.
- Multithreaded Architectures.
- What is scheduled dataflow architecture?
- Features of SDF architecture.
- Scalability of SDF
- Comparisons of SDF with Superscalar architecture.
- Summary

# Control flow vs. Dataflow Models of architecture

## Control Flow or von Neumann model:

In this type of architecture the execution of the instructions is controlled by the program counter and executes sequentially unless the sequence is changed by a jump, branch, interrupt or return.

## Dataflow Model:

The application program is translated into a dataflow graph (directed graph consisting of named nodes and arcs). Nodes represent instructions and arcs represent data dependencies among instructions.

The execution is data dependent as opposed to control driven. It eliminates the need for program counter.

# Multithreaded architectures

In a single threaded machine, program execution is defined by the processor state.

**Processor state:** Memory state (program memory, data memory, stack),  
Activity specifier (program counter and stack counter) and  
Register Context (set of registers).

**Context of thread:** Activity specifier and  
Register context.

(can hide memory latency, long I/O latency or can be used to interleave instructions on a cycle-by-cycle basis.)

# What is Scheduled Dataflow Architecture (SDF)?

It is a decoupled non-blocking multithreaded dataflow architecture.

**Blocking:** A thread can block during execution.  
(Memory accesses, cache miss, synchronization)  
(There could be too many context switch leading to smaller performance gain.)

**Non-Blocking:** A non-blocking thread proceeds to evaluation as soon as all the inputs are available. It completes execution without blocking the processor pipeline.  
(Basically thread switching is controlled by the compiler with the idea of creating new threads rather than blocking).  
(It could lead to creating of many small threads).

# What is Scheduled Dataflow Architecture (SDF)?

## Cont...

Memory Access times and I/O bandwidth have not improved compared to processor clock rate.  
The gap seems to be widening.

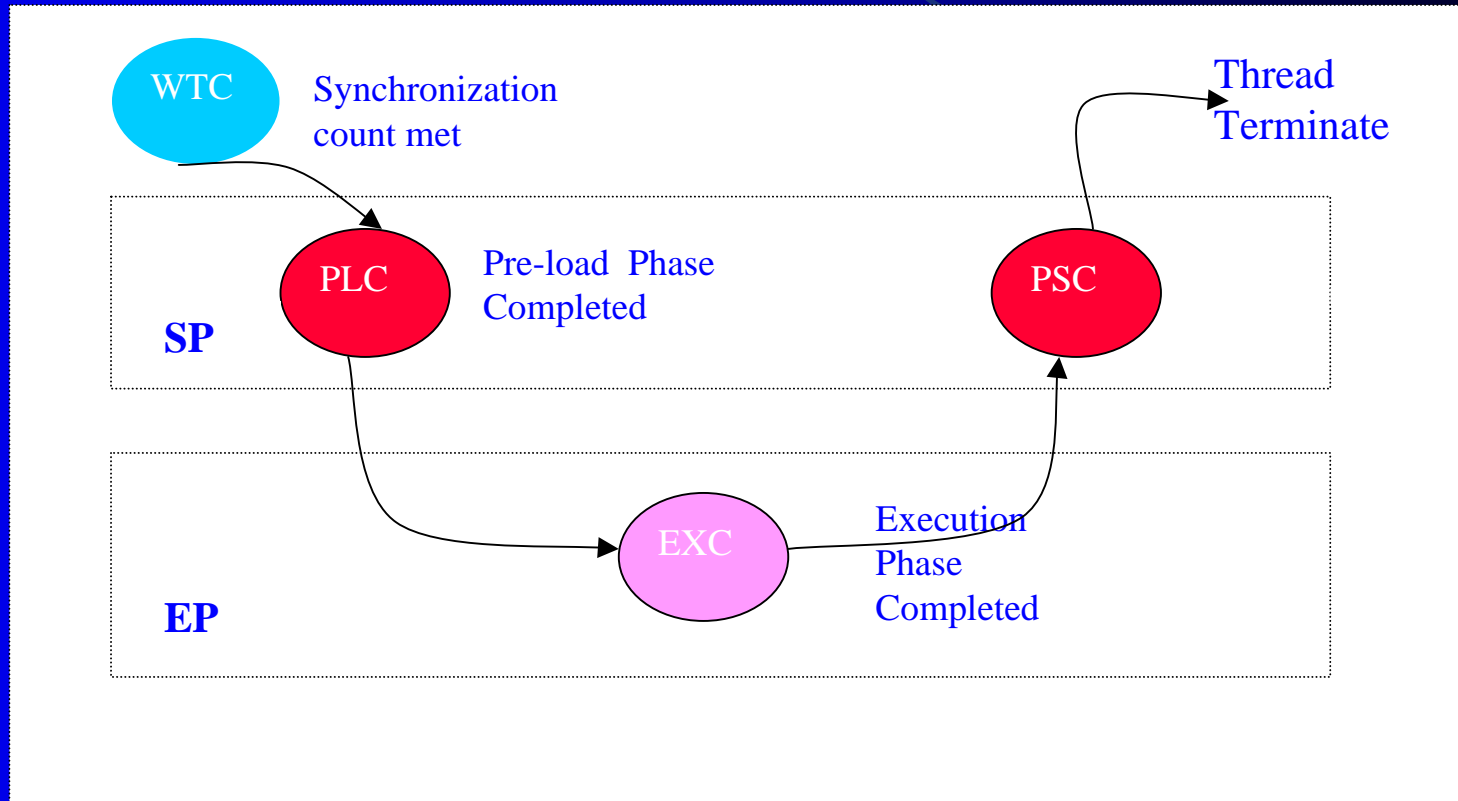
**Decoupled:** To separate the operand accesses from execution.  
Both instruction streams are executed on independent processing units

(Accessing processor AP and Execution processor EP)

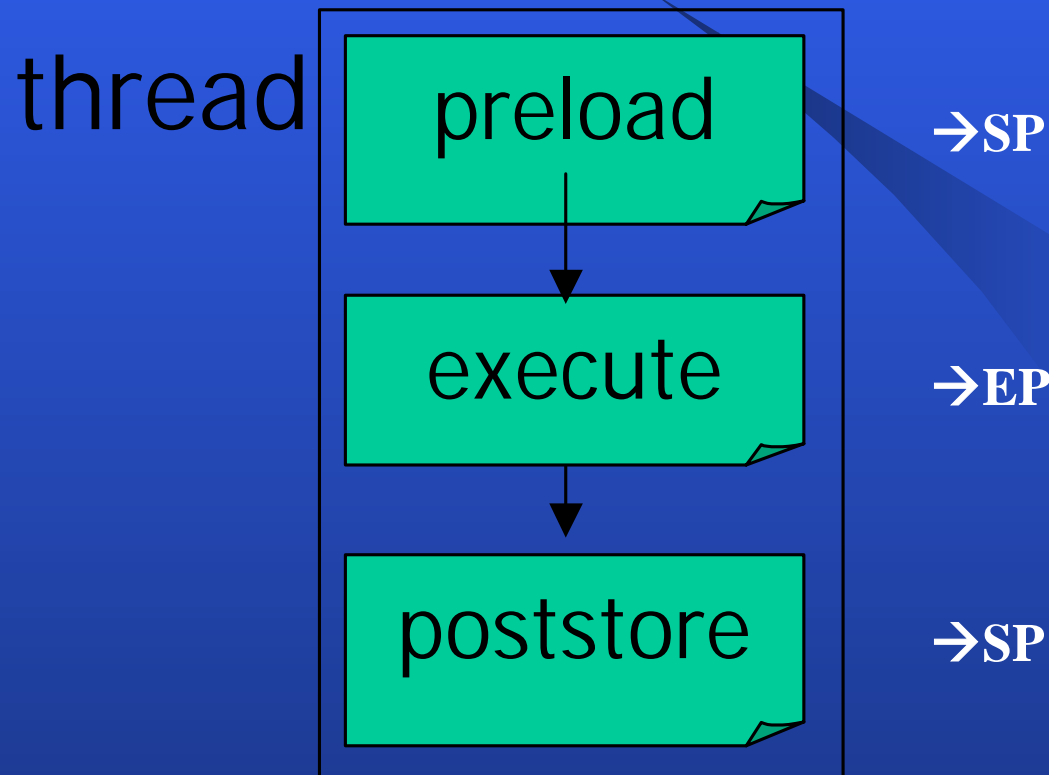
# Features of Scheduled Dataflow Architecture

1. **Instruction** driven rather than the token driven.
2. Instructions retain '**functional**' properties of dataflow.
3. Instructions are executed synchronously like control flow.
4. Memory accesses are completely decoupled from execution pipeline.
5. Non-blocking multithreaded model is used (A thread is enabled when all inputs are received. Once enabled, thread executes to completion without blocking or context switching).

# Continuation transition



# Thread code portion of SP and EP



**SP: Synchronization Process**

**EP: Execution Process**

# Representation of Dataflow in SDF

code main

```
LOAD RFP|2, R2      ; load A into R2
LOAD RFP|3, R3      ; load B into R3
LOAD RFP|4, R4      ; load X into R4
LOAD RFP|5, R5      ; load Y into R5
LOAD RFP|6, R6      ;load frame pointer for returning 1st result
LOAD RFP|7, R7      ;load frame offset for returning 1st result
LOAD RFP|8, R8      ;load frame pointer for returning 2nd result
LOAD RFP|9, R9      ;load frame offset for returning 2nd result
```

PUTR1 submain

FORKEP R1

STOP

```
submain: ADD  RR2,  R11,  R13      ; Compute A+B, Result in R11 and R13
          ADD  RR4,  R10      ; Compute X+Y, Result in R10
          SUB  RR4,  R12      ; Compute X-Y, Result in R12
          MULT RR10, R14      ; Compute (X+Y)*(A+B), Result in R14
          DIV  RR12, R15      ; Compute (X-Y)/(A+B), Result in R15
```

PUTR1 finemain

FORKSP R1

STOP

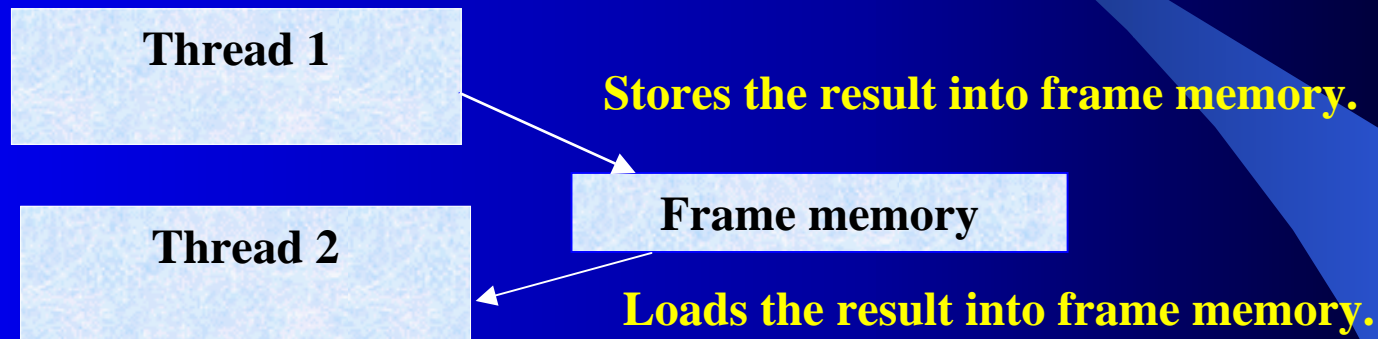
```
finemain: STORE R14,  R6|R7      ; Store first result
          STORE R15,  R8|R9      ; Store second result.
```

FFREE

STOP

## Two different ways of passing of data to a thread.

**1.** When a thread completes its execution, it stores the result into a frame memory. Next thread loads the data from the frame memory.

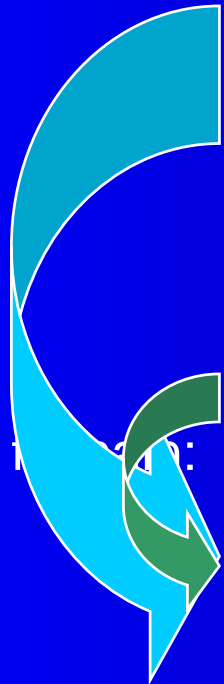


**2.** First thread stores the result into the second thread if the register set is available for the next thread.



# Usage of FALLOC with STORE & RALLOC with RSTORE

midmain: PUTR1 loop  
MOVE R1, R4  
PUTR1 5  
MOVE R1, R5  
**FALLOC RR4, R10**  
PUTR1 result  
MOVE R1, R4  
PUTR1 1  
MOVE R1, R5  
**FALLOC RR4, R11**



.....

**FALLOC**  
Result for the  
next thread is  
stored in the  
frame  
memory.

midmain: PUTR1 midloop  
MOVE R1, R4  
PUTR1 5  
MOVE R1, R5  
**RALLOC RR4, R10**  
PUTR1 midresult  
MOVE R1, R4  
PUTR1 1  
MOVE R1, R5  
**RALLOC RR4, R11**



finmain: .....

**RALLOC**  
Result for the  
next thread is  
stored in the  
registers .

## **Scalability of the SDF depends on the register sets**

**If we have more register sets, then we can have more threads spawned simultaneously. The results can be passed from one thread to the other without going through the frame memory.**

**When we have more threads simultaneously, the parallelism also improves. Thread granularity also can be improved and so the overall performance can be improved.**

**Having more threads simultaneously, improves the cache performance of the architecture since the thread data is handled on the registers avoiding of loading from the memory.**

**Since the hardware complexity is less, we can use that extra hardware savings to allocate more register sets.**

# SDF with multiple SPs and Eps

## Matrix multiply of different data sizes

Data size		Superscalar	SDF	Superscalar	SDF	Superscalar	SDF	Superscalar	SDF
		(cycles) 2INT ALU 1FP ALU	(cycles) 2SP 1EP	(cycles) 2INT ALU 2FP ALU	(cycles) 2SP 2EP	(cycles) 3INT ALU 2FP ALU	(cycles) 3SP 2EP	(cycles) 3INT ALU 3FP ALU	(cycles) 3SP 3EP
50*50	IO	1,890,104	1,504,297	1,890,104	860,782	1,867,200	756,707	1,867,200	<b>574242</b>
	OO	712,396		712,396		706,877		706,877	
100*100	IO	14,824,104	11,843,442	14,824,104	6,660,012	14,633,700	5,941,602	14,633,700	<b>4,402,772</b>
	OO	5,532,202		5,532,202		5,511,587		5,511,587	
150*150	IO	49,763,150	39,762,487	49,763,150	22,227,742	49,110,246	19,924,912	49,110,246	<b>14,819,482</b>
	OO	18,514,510		18,514,510		18,468,811		18,468,409	

Data size		Superscalar	SDF	Superscalar	SDF	Superscalar	SDF	Superscalar	SDF
		(cycles) 4INT ALU 3FP ALU	(cycles) 4SP 3EP	(cycles) 4INT ALU 4FP ALU	(cycles) 4SP 4EP	(cycles) 5INT ALU 4FP ALU	(cycles) 5SP 4EP	(cycles) 5INT ALU 5FP ALU	(cycles) 5SP 5EP
50*50	IO	1,867,200	<b>507,197</b>	1,867,200	<b>430,957</b>	1,867,200	<b>381,247</b>	1,867,200	<b>345,027</b>
	OO	680,321		680,321		680,321		680,321	
100*100	IO	14,633,700	<b>3,970,682</b>	14,633,700	<b>3,330,992</b>	14,633,700	<b>2,982,702</b>	14,633,700	<b>2,665,472</b>
	OO	5,306,381		5,306,381		5,306,380		5,306,380	
150*150	IO	49,110,246	<b>13,308,457</b>	49,110,246	<b>11,115,592</b>	49,110,246	<b>9,990,607</b>	49,110,246	<b>8,894,002</b>
	OO	17,782,453		17,782,453		17,782,453		17,782,453	

## Comparison of SDF with VLIW Matrix multiply program

<b>Matrix Multiplication</b>					
<b>Data Size</b>	<b>SDF</b>	<b>Trimaran</b>	<b>TMS 'C6000</b>	<b>SDF/Trimaran</b>	<b>SDF/TMX 'C6000</b>
50*50	430957	331910	1033698	1.29841523	0.416908033
100*100	3330992	2323760	16199926	1.43344924	0.205617729
150*150	11115592	4959204	86942144	2.24140648	0.127850447

**Trimaran uses a well optimized compiler as well as loop unrolling of program many times and so it certainly improves performance.**

# Limited Register sets usage and the thread Allocation of matrix multiply program

<b>N=50</b>	<b>1 Thread</b>	<b>2 Threads</b>	<b>5 Threads</b>
1SP 1EP	4985034 (3 Reg.Set)	2381565 (12 Reg.Set)	2330723 (70Reg.Set)
2SP 2EP	4985911 ( " )	1246765 ( " )	1166734 ( " )
3SP 3EP	4985011 ( " )	916006 ( " )	778910 ( " )
4SP 4EP	4985011 ( " )	810961 ( " )	585357 ( " )
<b>N=100</b>	<b>1 Thread</b>	<b>2 Threads</b>	<b>5 Threads</b>
1SP 1EP	38929984 (3 Reg.Set)	18581732 (12 Reg.Set)	18369157 (70Reg.Set)
2SP 2EP	38929961 ( " )	9582703 ( " )	9186942 ( " )
3SP 3EP	38929961 ( " )	7069808 ( " )	6126821 ( " )
4SP 4EP	38929961 ( " )	6164461 ( " )	4597262 ( " )
<b>N=150</b>	<b>1 Thread</b>	<b>2 Threads</b>	<b>5 Threads</b>
1SP 1EP	130334934 (3 Reg.Set)	62194407 (12 Reg.Set)	61689117 (64Reg.Set)
2SP 2EP		31862481 ( " )	30847669 ( " )
3SP 3EP		23473169 ( " )	20568612 ( " )
4SP 4EP		20445811 ( " )	15430183 ( " )

# Summary and conclusion

- SDF architecture separates memory accesses and execution neatly
  - because of the non-blocking multithreaded model
- The non-blocking model allows the pipes to execute independently pre-load/post-store/Execute portions of a thread
  - fewer context switches(possibly more threads)
- SDF architecture outperforms MIPS, superscalar and VLIW .
  - because of multithreaded model
- SDF scales better with more functional units
  - because of multithreaded model
- The SDF architecture is simpler since it does not need complex hardware for detecting data hazards, register renaming or out-of-order instruction execution
  - because of the functional nature of dataflow instructions