A Simple Non-Blocking Multithread Architecture

Krishna Kavi

(with H.-S. Kim and A.R. Hurson)

Department of Electrical and Computer Engineering

The University of Alabama in Huntsville

kavi@ece.uah.edu

http://crash.eb.uah.edu/~kavi

Jump to first page

Blocking vs Non-Blocking Models

Non-Blocking Models:

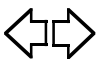
A thread, once scheduled for execution cannot be stopped or pre-empted until the thread completes execution

May need to create more threads

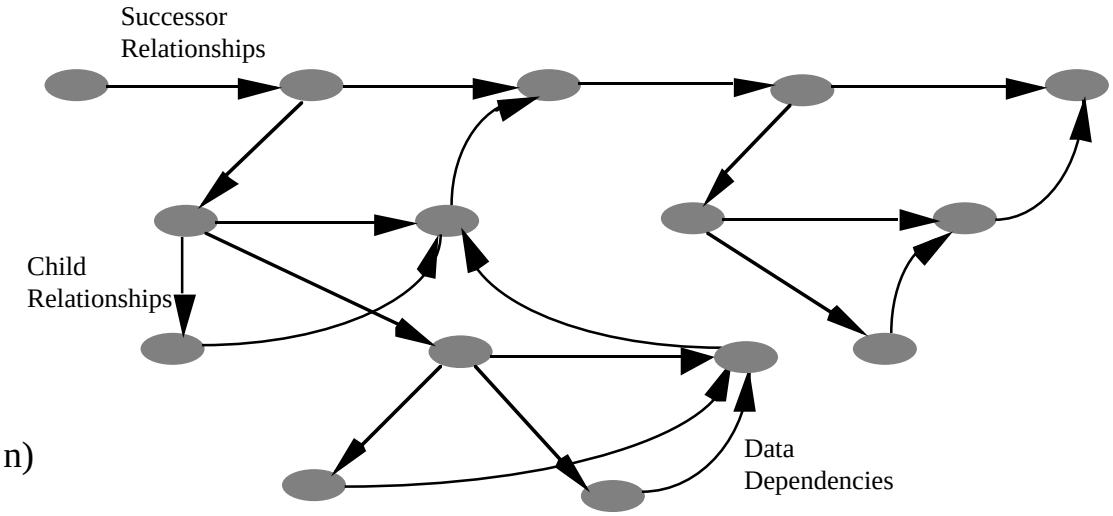
Blocking Models:

A thread can be stopped, blocked on a resource, pre-empted by another thread and subsequently resumed for execution

May require more context switches



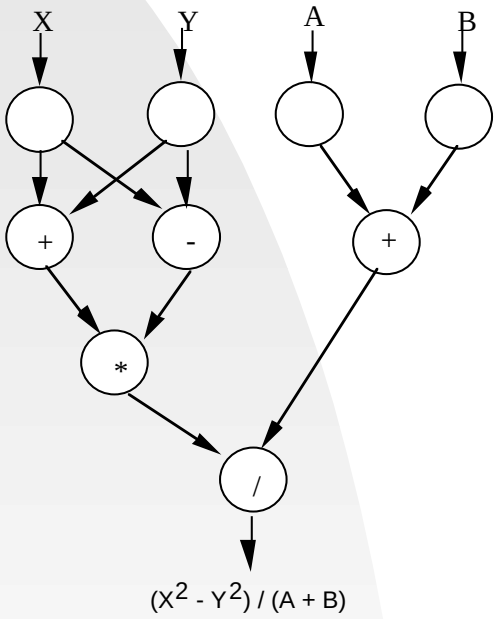
A Non-Blocking Program Example




```
thread fib (cont int k, int n)
    {          if (n<2)
                    send_argument (k, n)
               else{
                    cont int x, y;
                    spawn_next sum (k, ?x, ?y);  /* create a successor thread
                    spawn fib (x, n-1);     /* fork a child thread
                    spawn fib (y, n-2);     /* fork a child thread
                    }

               thread sum (cont int k, int x, int y)
                    send_argument (k, x+y);
                              /* return results to parent's successor
```

What is a dataflow architecture?



Consider The Following Example

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

Dataflow

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

Conventional

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

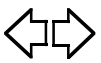
Dataflow Multithreading

In pure dataflow, each instruction can be viewed as an independent thread.

An instruction is enabled only when its operands are made available by predecessor instructions

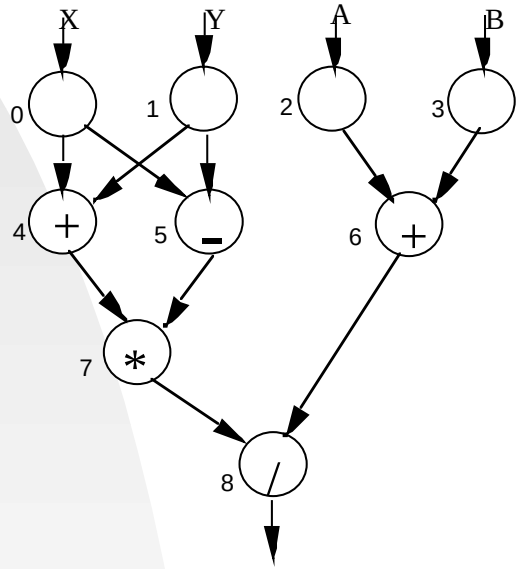
The “context” or “continuation” of an instruction (thread) is used for forwarding operands to instructions.

Consider Explicit Token Store Dataflow Model.



Explicit Token Store Architecture (ETS)

0: In, 8, 4_L, 5_L
1: In, 7, 4_R, 5_R
2: In, 6, 6_L
3: In, 5, 6_R
4: +, 4, 7_L
5: - , 3, 7_R
6: +, 2, 8_R
7: *, 1, 8_L
8: /, 0, Out



Consider the instruction format

Opcode	Offset (R)	Dest-1 and Port	Dest-2 and Port

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.

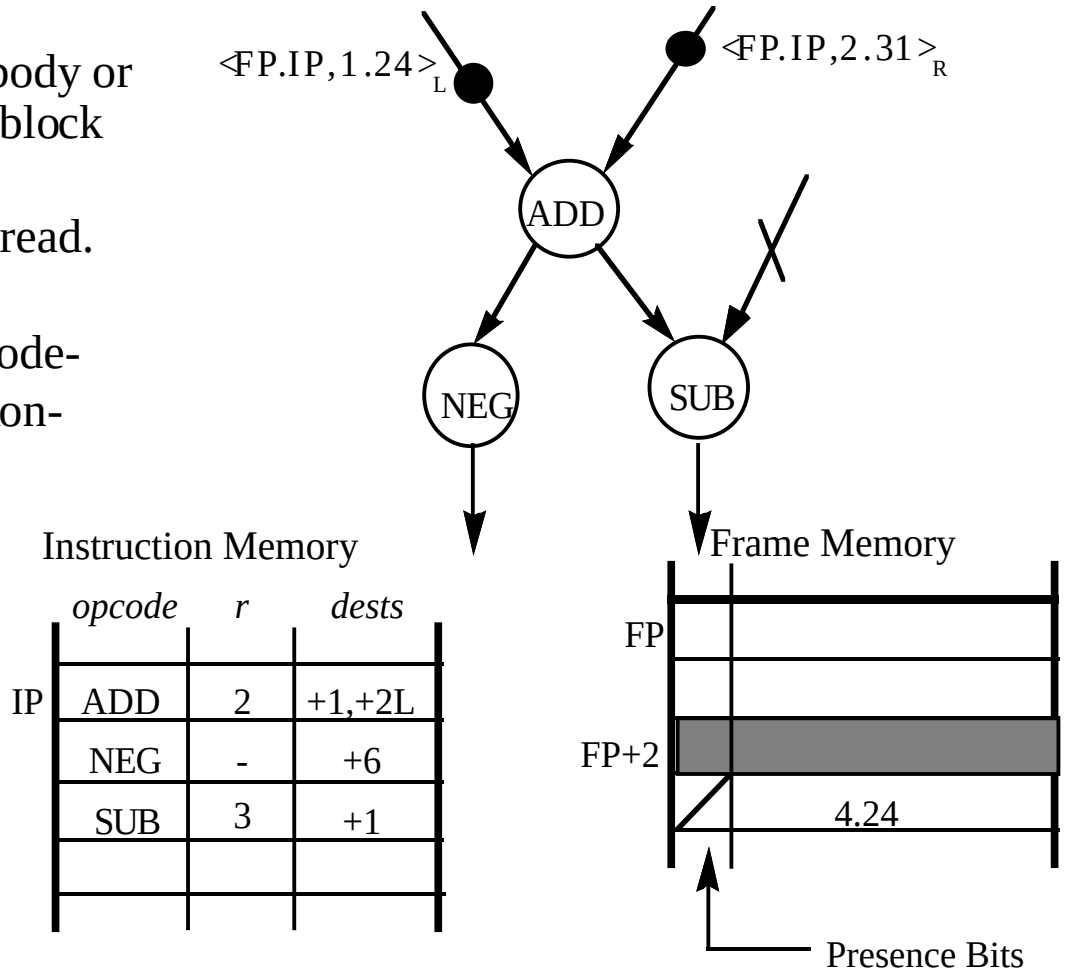
PE#	Context **FP**	Instruction **IP**	port	Data Value

ETS Continued

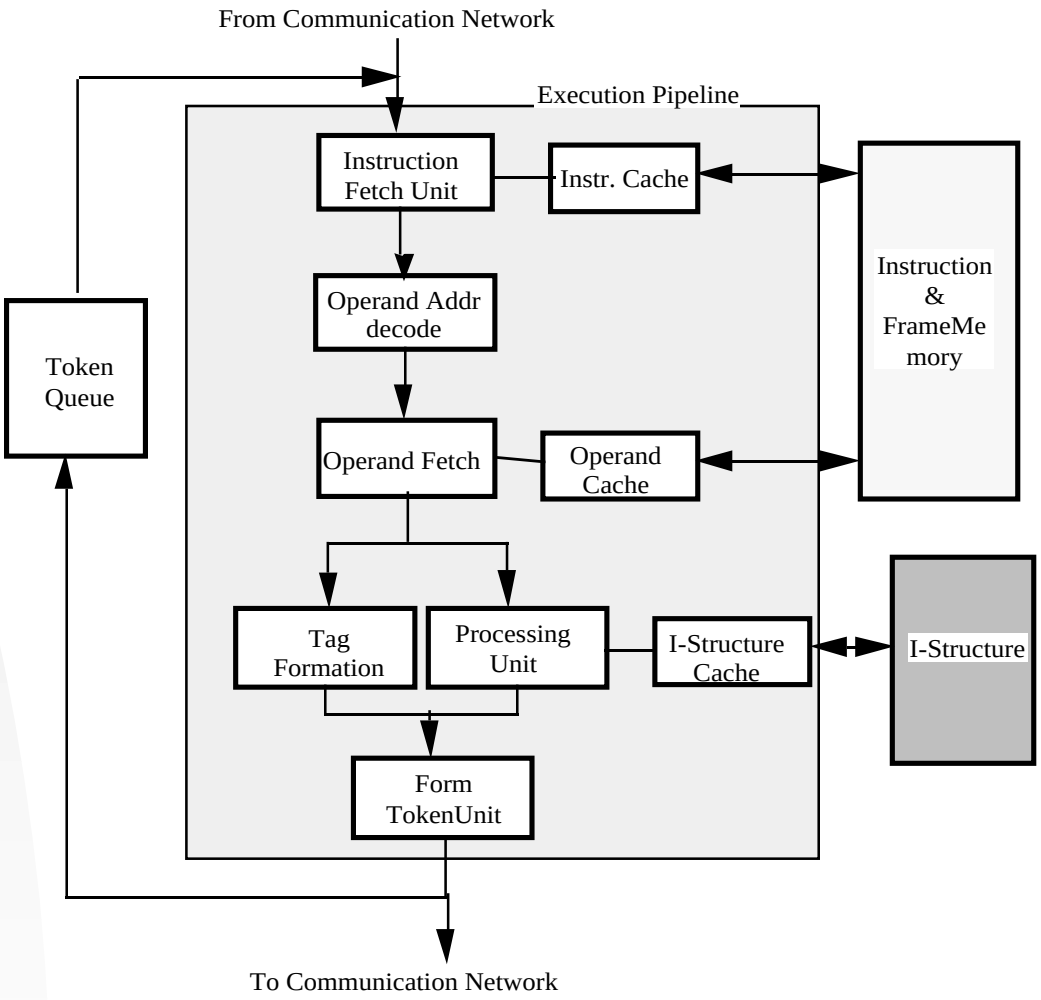
ETS Code Blocks. -- A loop body or a function is treated as a code block

Can be viewed as a "coarser-grained" thread.

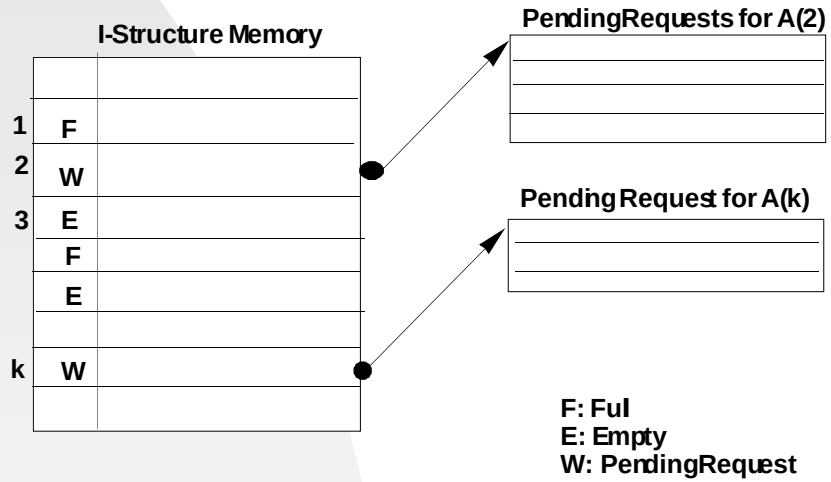
In actual implementations, a code-block may consist of several non-blocking threads



An implementation of ETS with Caches



What are I-structures?



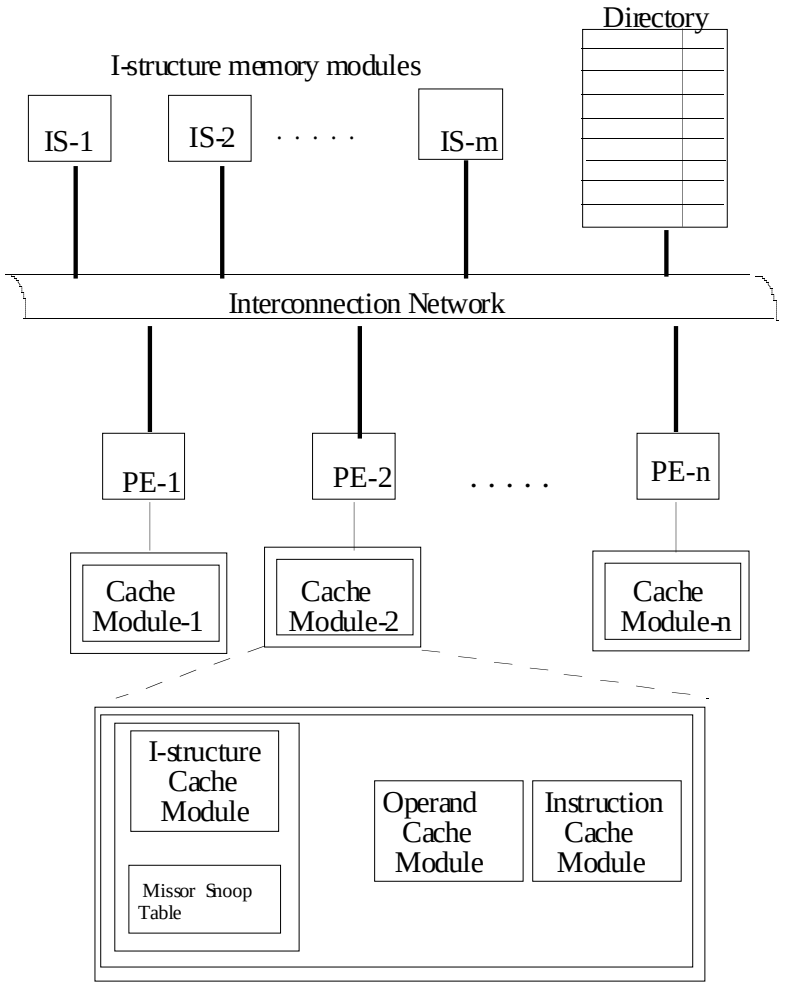
Used to store Arrays (or other data structures)

Single assignment is still maintained

Instructions needed:

- Allocate (A, N)
- I-Store (A, I, Value)
- I-Fetch (A, I)

A multiprocessor environment for ETS



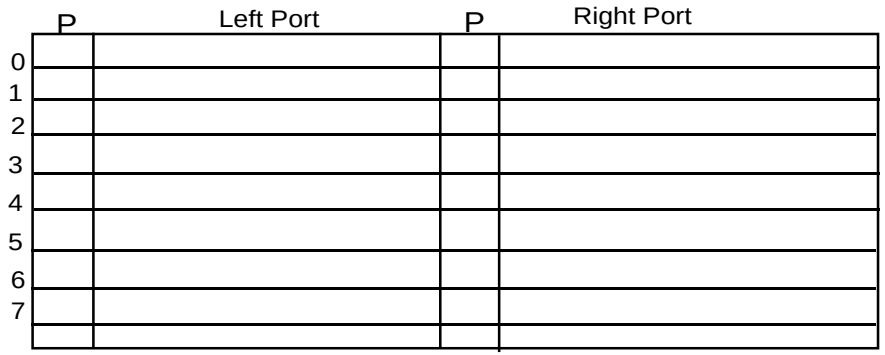
Synchronous execution of dataflow

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

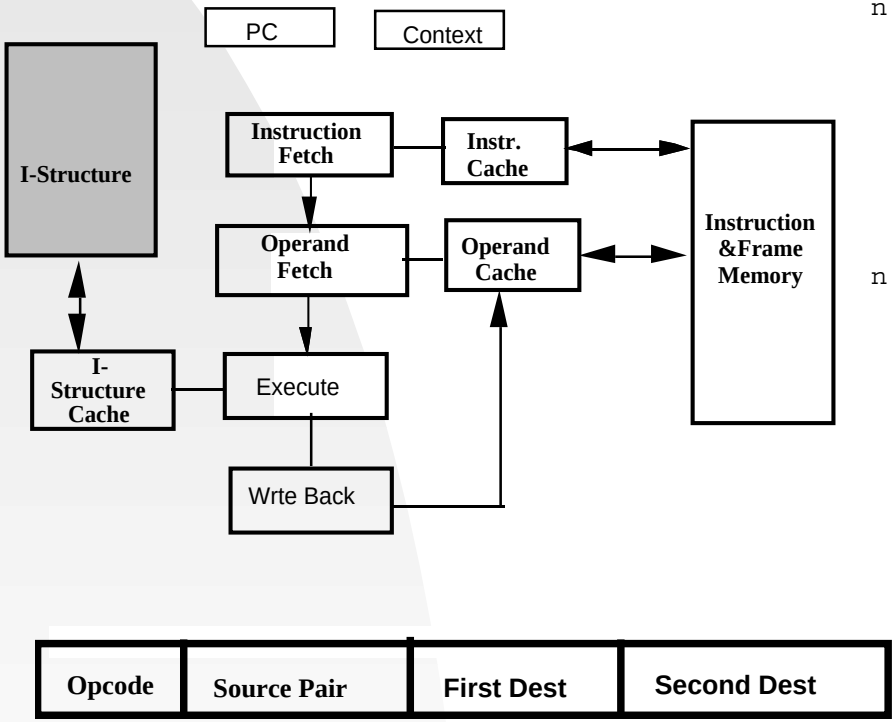
How can we execute dataflow instructions synchronously -- requiring only one cycle per instruction? (That is, make them *instruction-driven*)

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



Scheduled Dataflow Architecture



- 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.

Digression: Decoupled memory access

Separate processor to handle all memory accesses

The earliest suggestion by J.E. Smith – DAE architecture (1982)

More recent implementations include

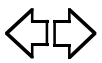
RHAMMA -- from University of Karlsruhe

and PL/PS --- by us

Others have used two separate processors:

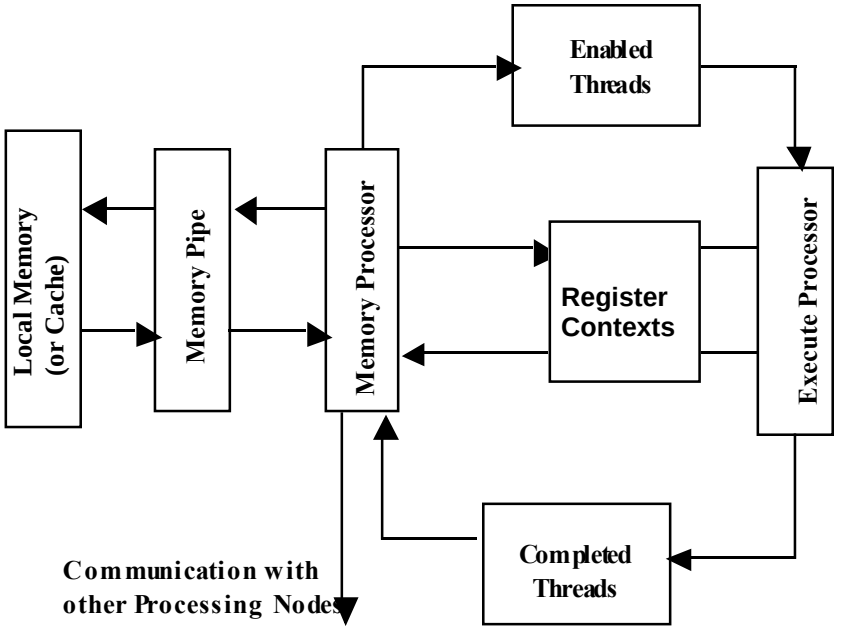
One processor for thread scheduling

One processor for thread execution



Pre-Load/Post-Store (PL/PS) 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



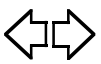
A simple example

```
LD      F0, 0(R1)
LD      F6, -8(R1)
MULTD   F0, F0, F2
MULTD   F6, F6, F2
LD      F4, 0(R2)
LD      F8, -8(R2)
ADDD    F0, F0, F4
ADDD    F6, F6, F8
SUBI    R2, R2, 16
SUBI    R1, R1, 16
SD      8(R2), F0
BNEZ    R1, LOOP
SD      0(R2), F6
```

Conventional

```
LD      F0, 0(R1)
LD      F6, -8(R1)
LD      F4, 0(R2
LD      F8, -8(R2)
MULTD   F0, F0, F2
MULTD   F6, F6, F2
SUBI    R2, R2, 16
SUBI    R1, R1, 16
ADDD    F0, F0, F4
ADDD    F6, F6, F8
SD      8(R2), F0
SD      0(R2), F6
```

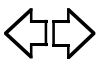
New Architecture



Features of PL/PS

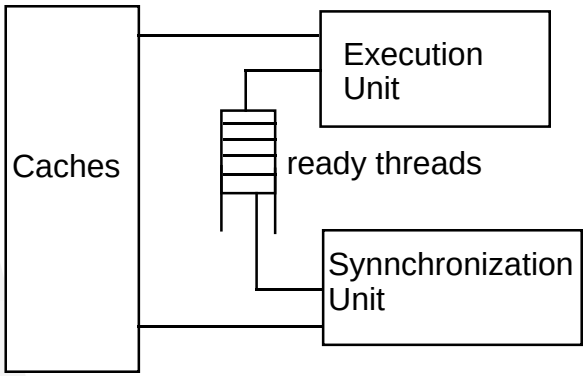
- Multiple hardware contexts
- No pipeline bubbles due to cache misses
- Overlapped execution of threads
- Opportunities for better data placement and prefetching
- Fine-grained threads -- A limitation?
- Multiple hardware contexts add to hardware complexity

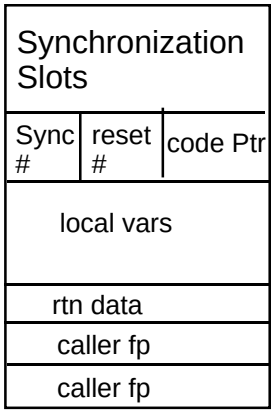
If 35% of instructions are memory access instructions, PL/PS can achieve 35% increase in performance with sufficient thread parallelism and completely mask memory access delays!



Hybrid Architecutres

Dataflow like scheduling at thread level
Threads are Coarse Grained
Threads are comprised of conventional control flow instruction





Activation frame

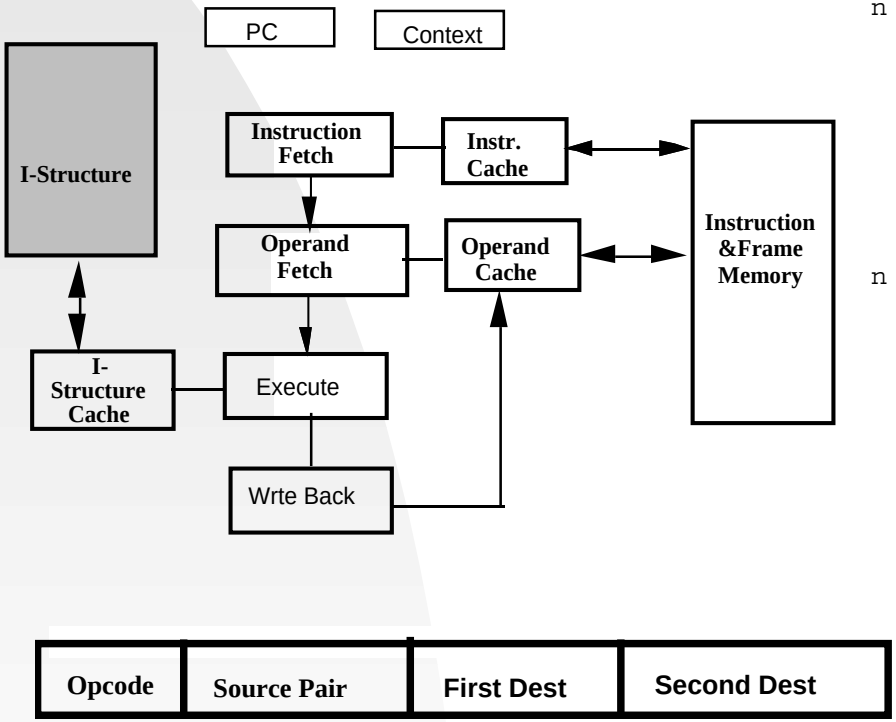
Earth Hybrid Dataflow Architecture

Jump to first page

Back to dataflow architectures: Scheduled Dataflow

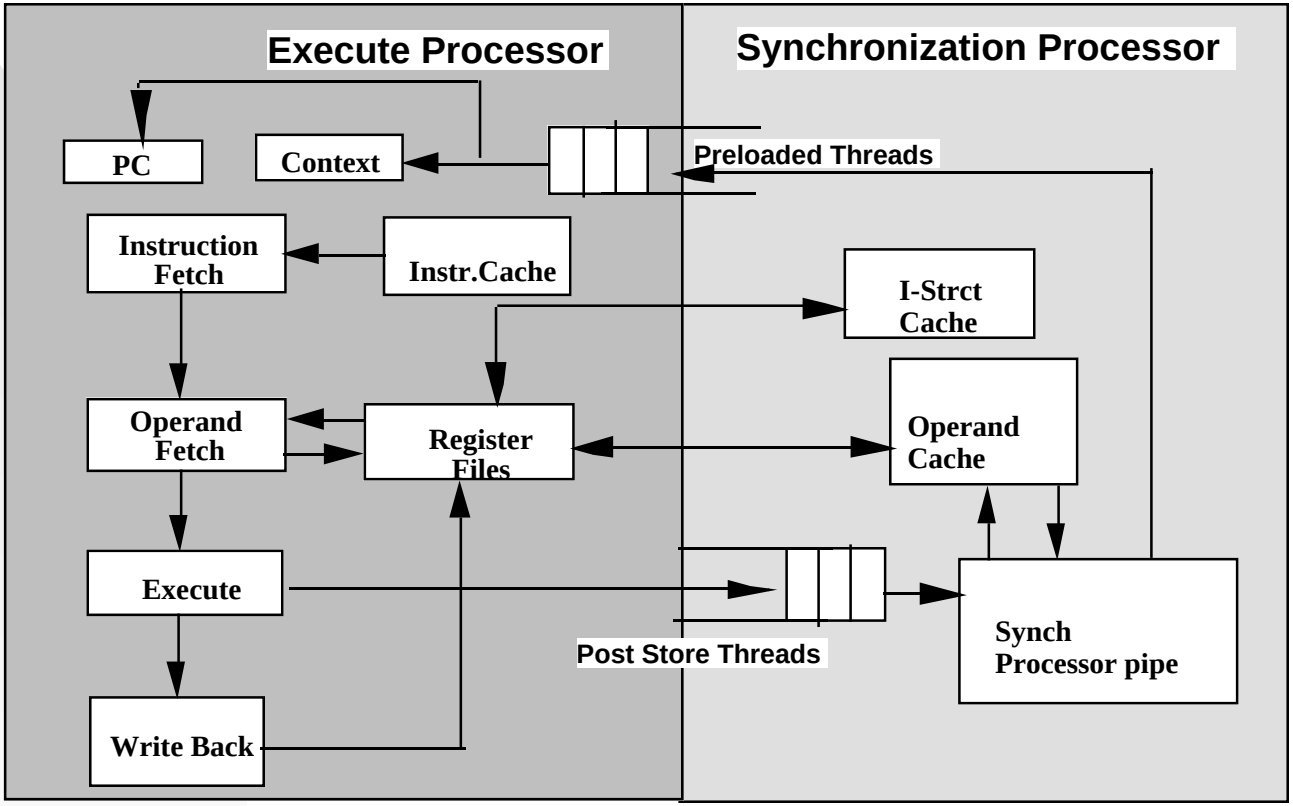
- Brings dataflow closer to conventional RISC architecture
- Utilizes Decoupled processors to eliminate pipeline bubbles on cache misses -- combines Preload/post-store with dataflow
- Eliminates WAR and WAW dependencies in pipelines
  The result of using dataflow execution
- Uses Non-blocking Multithreaded model

Scheduled Dataflow Architecture

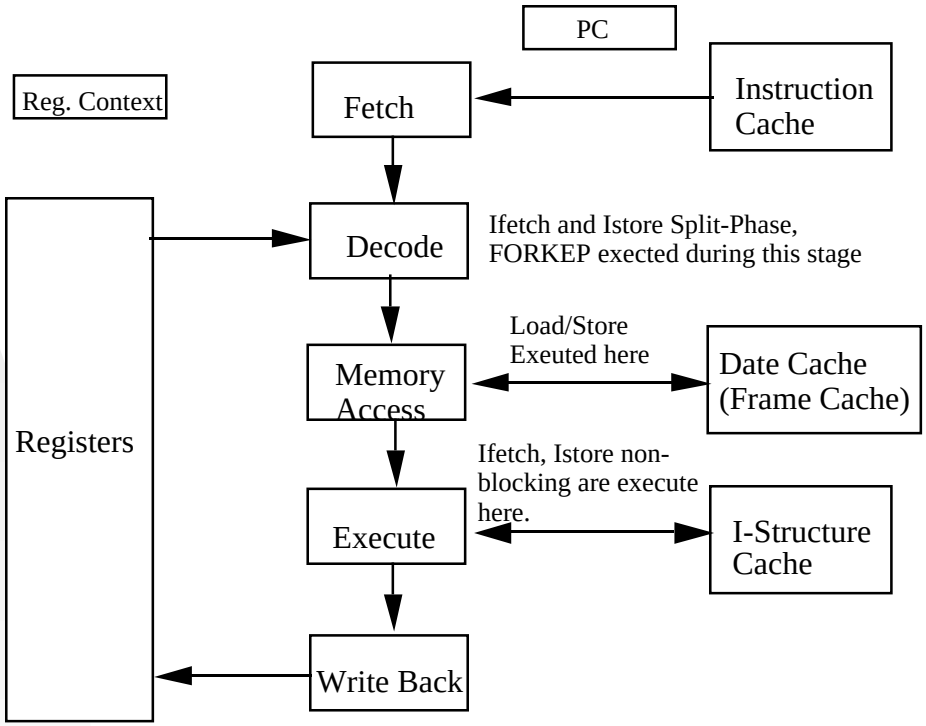


- Each instruction is associated with a pair of source registers.
- Predecessor instructions store their results in these registers.
- An instruction is not enabled immediately when the two source registers are loaded.
- Instructions are scheduled similar to conventional processors.
- However, instructions retain functional properties.

Decoupled processors for Scheduled Datafllow

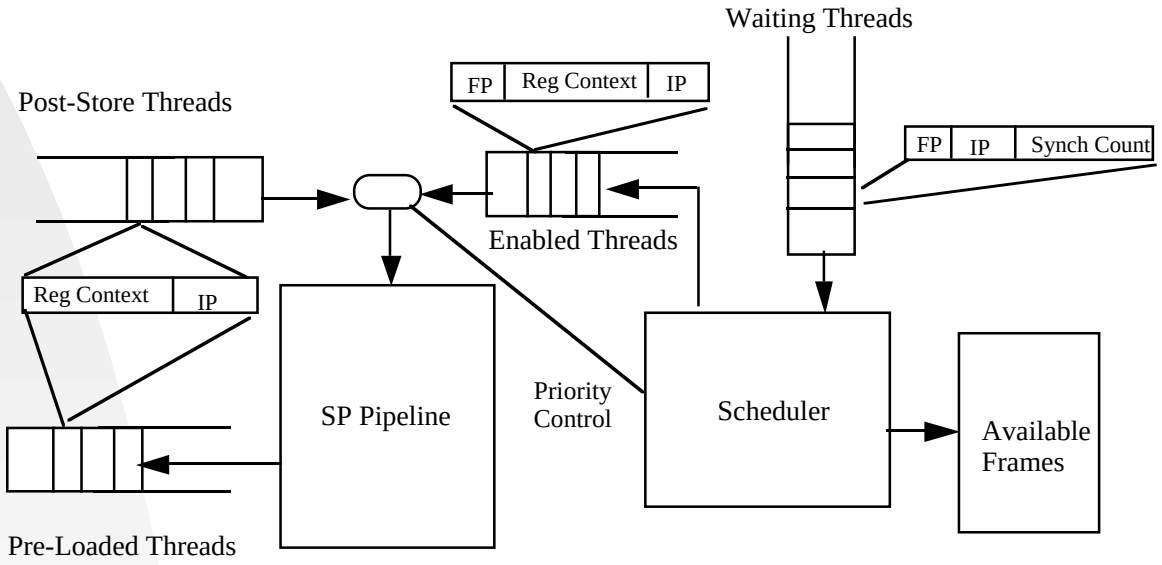


Synchronization Processor Design



Synchronization Processor Pipeline

Synchronization Processor Design



Preliminary performance comparisons

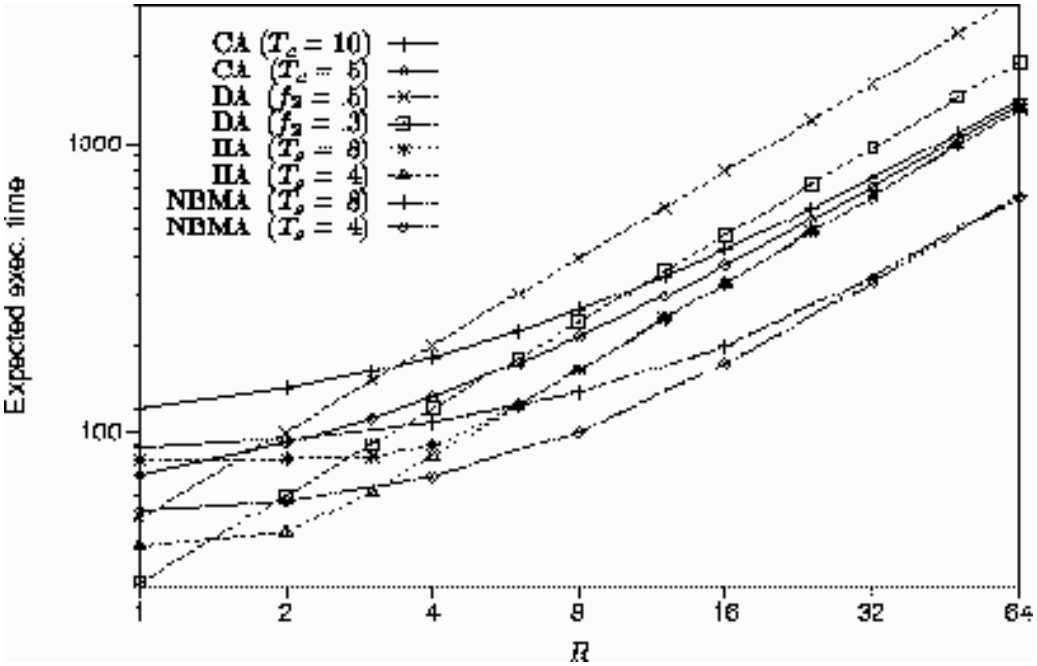
- Monte Carlo simulations using simple models for, Scheduled Dataflow, ETS, conventional RISC processors and Hybrid dataflow/control-flow architectures.
- Some of the parameters are based on published data (% load/stores, avg memory latency, cache miss rates, context switching overhead).
- Some parameters are based on simple programs coded in our architecture (e.g., matrix multiply, livermore loops).
- Some parameters are based on guesswork.

Thread Granularity

Except for very fine grained threads,Scheduled Dataflow outperform other architectures
Moderate granularity (8-16 instructions) is sufficient.

ETS is always fine-grained

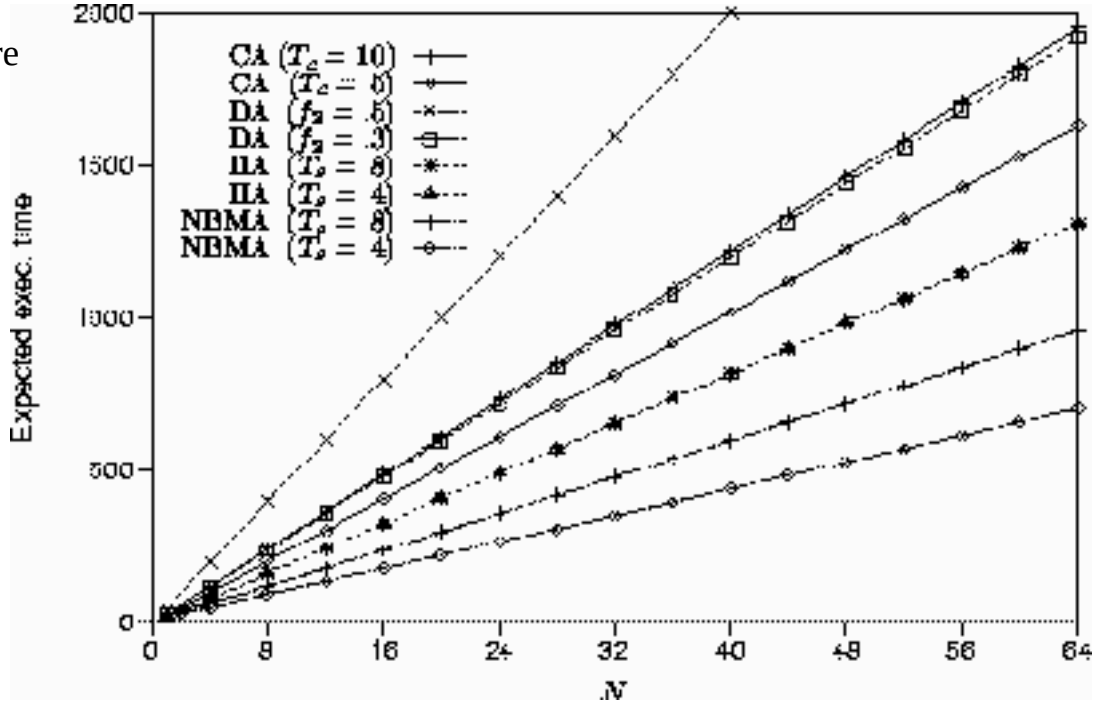
Earth (HA) does not decouple memory accesses



CA: Conventional Architecture
DA: ETS like Dataflow
HA: Earth Like Hybrid
NBMA: Scheduled Dataflow

Effect Of Thread Level Parallelism

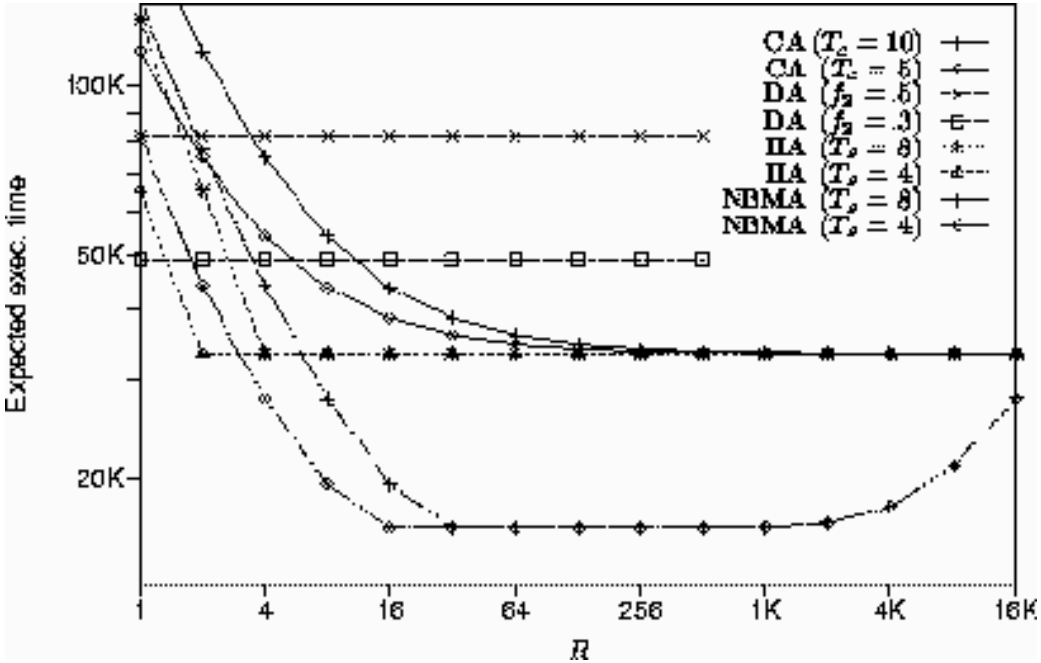
CA: Conventional Architecture
DA: ETS like Dataflow
HA: Earth Like Hybrid
NBMA: Scheduled Dataflow



More parallelism in Scheduled Dataflow means more opportunities for overlap between Synchronization processor and Execution Processor

Thread Granularity Vs Thread Parallelism

CA: Conventional Architecture
DA: ETS like Dataflow
HA: Earth Like Hybrid
NBMA: Scheduled Dataflow



For the same total workload, best performance is achieved when there is a balance between thread granularity and thread parallelism.

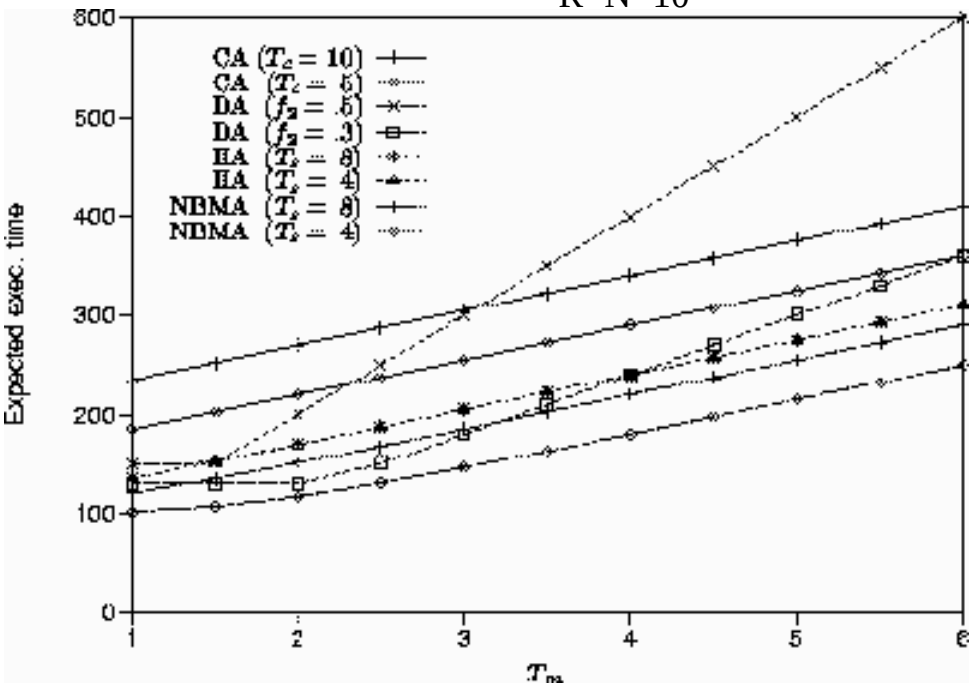
ETS --always fine grained
Scheduled dataflow performs well for moderate granularity

Effect Of Memory Access Time

Tm includes cache misses and miss penalties.

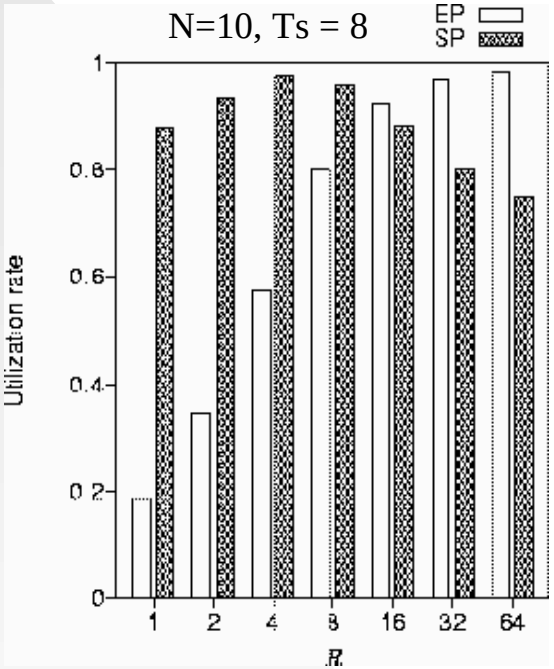
Scheduled dataflow (and Hybrid) tolerate longer memory access times better.

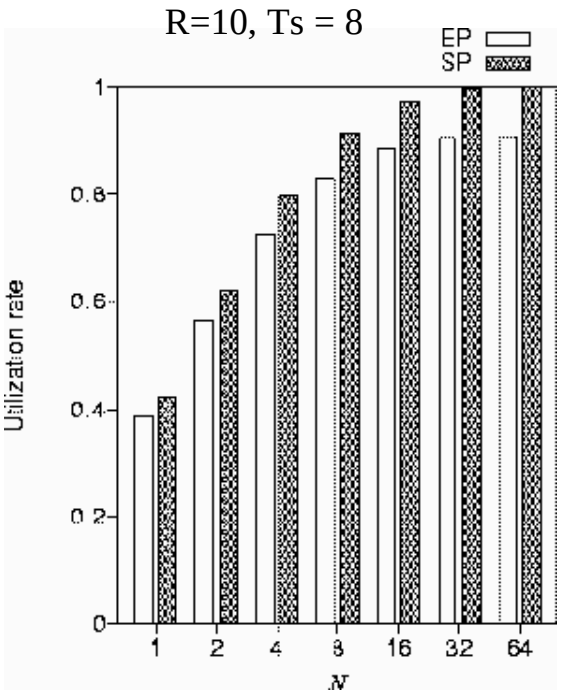
R=N=10



CA: Conventional Architecture
DA: ETS like Dataflow
HA: Earth Like Hybrid
NBMA: Scheduled Dataflow

Utilization Of EP and SP





Except when very fine grained threads, Synchronization Processor is not a bottleneck.

For moderate sized threads, there is a balanced utilization of the two processors

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
  - Using MIDC compiler from Colorado State Univ
- 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