MODULE VI

MODULE VI

- Multithreaded and data flow architectures
  - **Latency hiding techniques**
  - Principles of multithreading
    - Multithreading Issues and Solutions
    - Multiple context Processors
  - Fine-grain Multicomputer
    - Fine-grain Parallelism
  - Dataflow and hybrid architecture

Multithreaded and Data Flow Architectures

- **Latency Hiding Techniques**
  - The processor speed has been increasing at a much faster rate than memory speeds
  - Any scalable multiprocessor or large-scale multicomputer must rely on the use of latency-reducing, -tolerating, or —hiding mechanisms
  - Four latency-hiding mechanisms for enhancing scalability and programmability

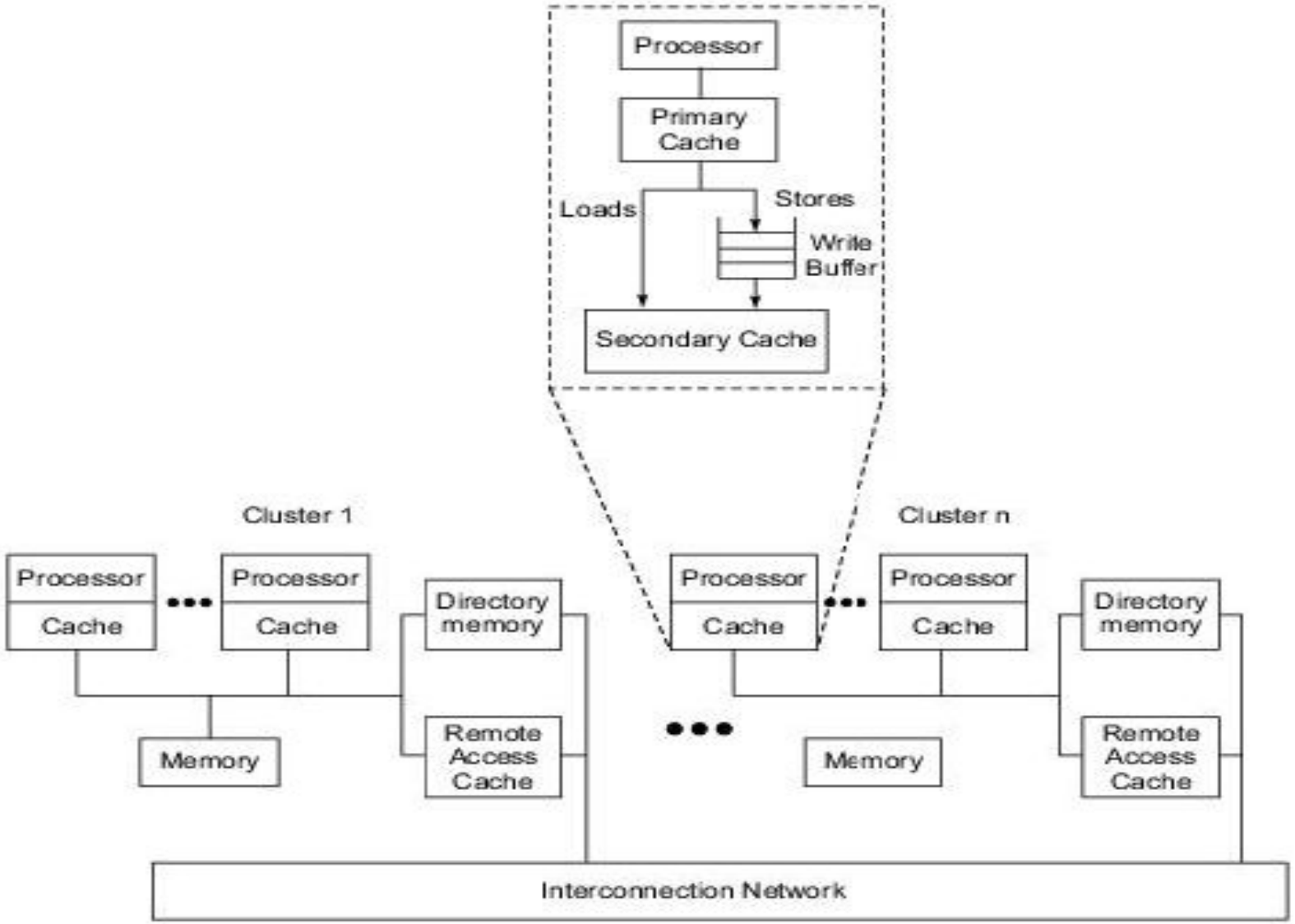
- Latency hiding can be accomplished through :
  - **Using pre-fetching techniques** (bring instructions or data close to the processor before they are actually needed)
  - **Using coherent Caches** (supported by hardware to reduce cache misses)
  - **Using relaxed memory consistency** (allowing buffering and pipelining of memory references)
  - **Using multiple Context** (processor to switch from one contest to another when a long-latency operation is encountered)

Shared virtual memory

- Single-address-space multiprocessors/multicomputers must use shared virtual memory

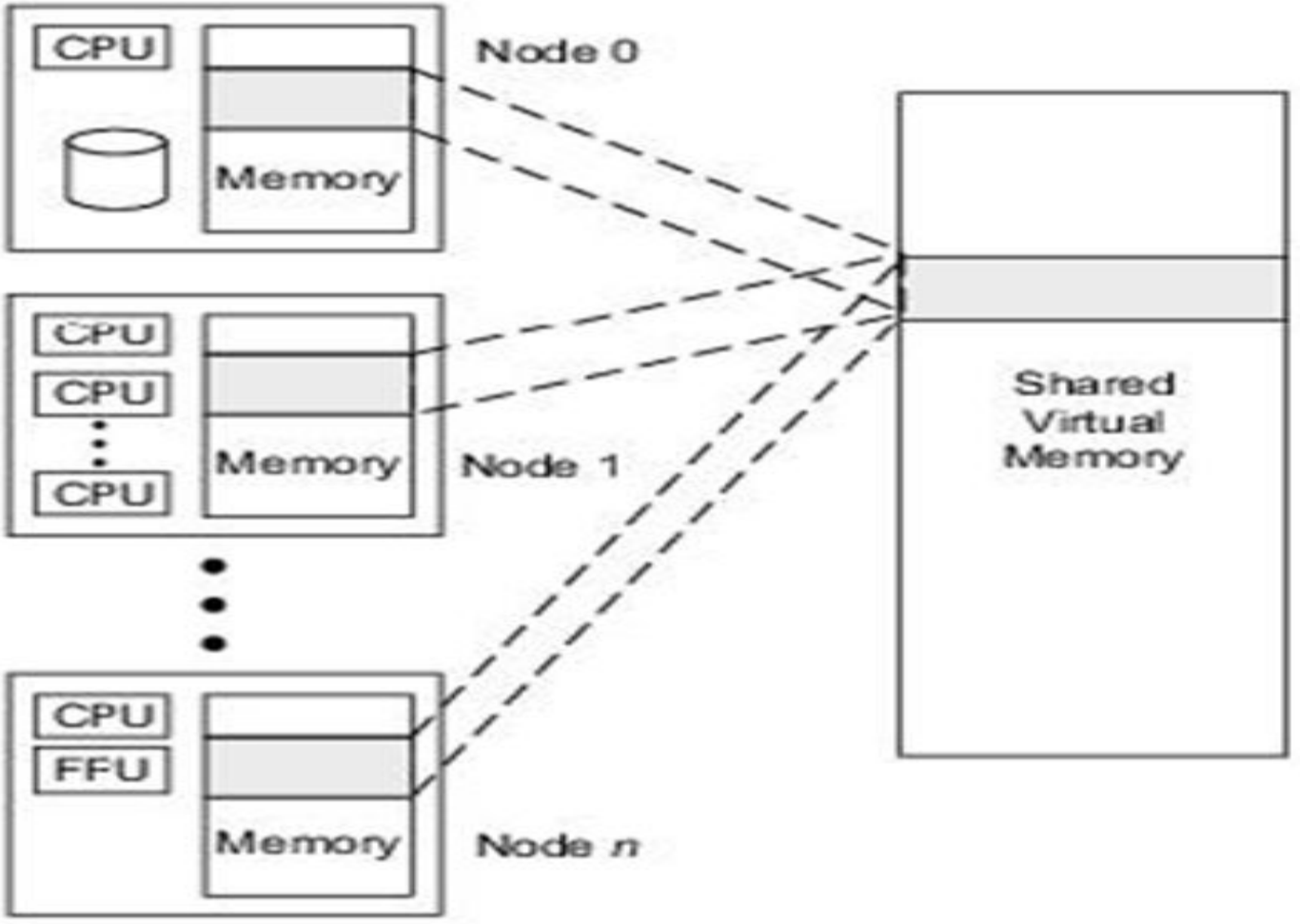
- **The architecture environment**

Fig : A scalable coherent cache multiprocessor with distributed shared memory modeled after the Stanford



- Two levels of local cache
- Write Buffers
- The SVM Concept : A global virtual address space is shared among processors residing at a large number of loosely coupled processing nodes
- Shared Virtual Memory

Fig :The concept of distributed shared memory with a global virtual address space shared among all processors on loosely coupled processing nodes in a massively parallel architecture



(a) Distributed shared memory

- **Page Swapping :**

Representative SVM Research Systems

System and Developer	*Implementation and Structure*	*Coherence Semantics and Protocols*	*Special Mechanics for Performance and Synchronization*
Stanford Dash (Lenoski, Laudon, Gharachorloo, Gupta, and Hennessy, 1988–).	Mesh-connected network of Silicon Graphics 4D/340 workstations with added hardware for coherent caches and prefetching.	Release memory consistency with write-invalidate protocol.	Relaxed coherence, prefetching, and queued locks for synchronization.
Yale Linda (Carriero and Gelernter, 1982–).	Software-implemented system based on the concepts of tuple space with access functions to achieve coherence via virtual memory management.	Coherence varied with environment; hashing used in associative search; no mutable data.	Linda could be implemented for many languages and machines using C-Linda or Fortran-Linda interfaces.
CMU Plus (Bisiani and Ravishankar, 1988–).	A hardware implementation using MC 88000, Caltech mesh, and Plus kernel.	Used processor consistency, nondemand write-update coherence, delayed operations.	Pages for sharing, words for coherence, complex synchronization instructions.
Princeton Shiva (Li and Schaefer, 1988).	Software-based system for Intel iPSC/2 with a Shiva/native operating system.	Sequential consistency, write-invalidate protocol, 4-Kbyte page swapping.	Used data structure compaction, messages for semaphores and signal-wait, distributed memory as backing store.

Prefetching Techniques

- **Binding Prefetch**
- nonbinding prefetching also brings the data close to the processor
- since the value will become stale if another processor modifies the same location during the interval between prefetch and reference.

Prefetching Techniques

- **Non- Binding Prefetch**

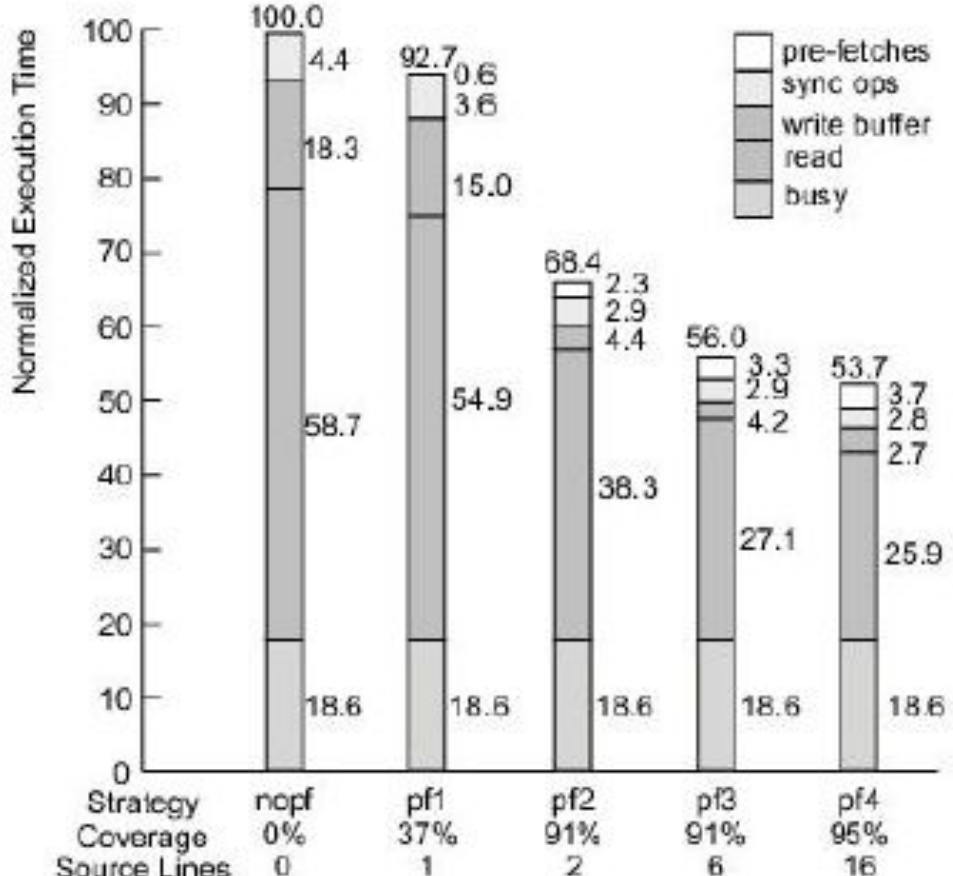
- brings the data close to the processor, but the data remains visible to the cache coherence protocol
- Software prefetching
  - explicit prefetch instructions are issued
- Hardware prefetching
  - Hardware prefetch at runtime
  - It provide better dynamic information
  - No instruction overhead to issue prefetches

Benefits of Prefetching

- when a prefetch is issued early enough in the code so that the line is already in the cache by the time it is referenced.

Prefetching Techniques

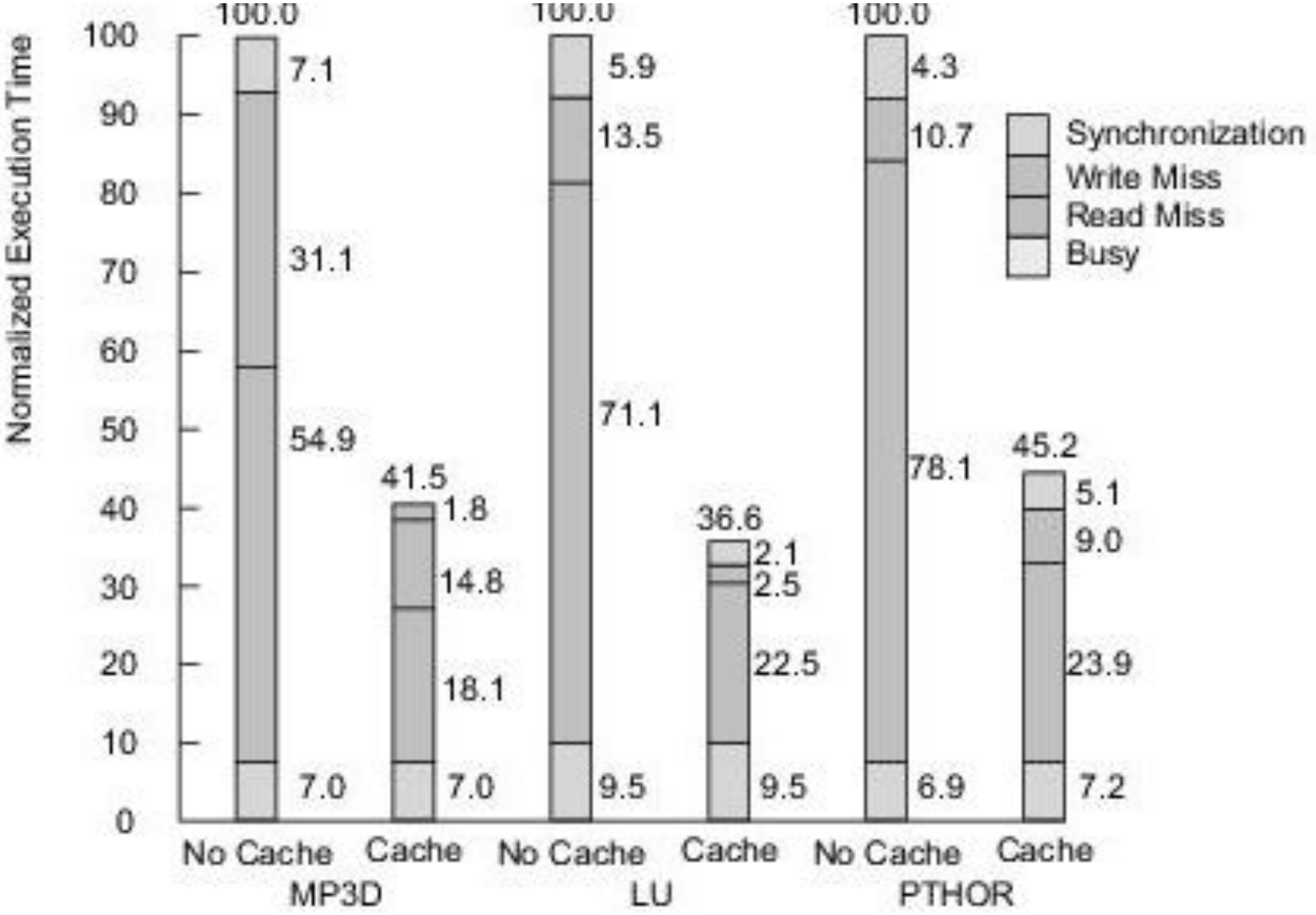
- Benefits of Prefetching
- Benchmark Result



Distributed Coherent Caches

- Coherence problem is easily solved for small bus-based multiprocessors through the use of snoop cache coherence protocols
- **Benefits of Cacheing** : the cacheing of shared read-write data provided substantial gains in performance,

Distributed Coherent Caches

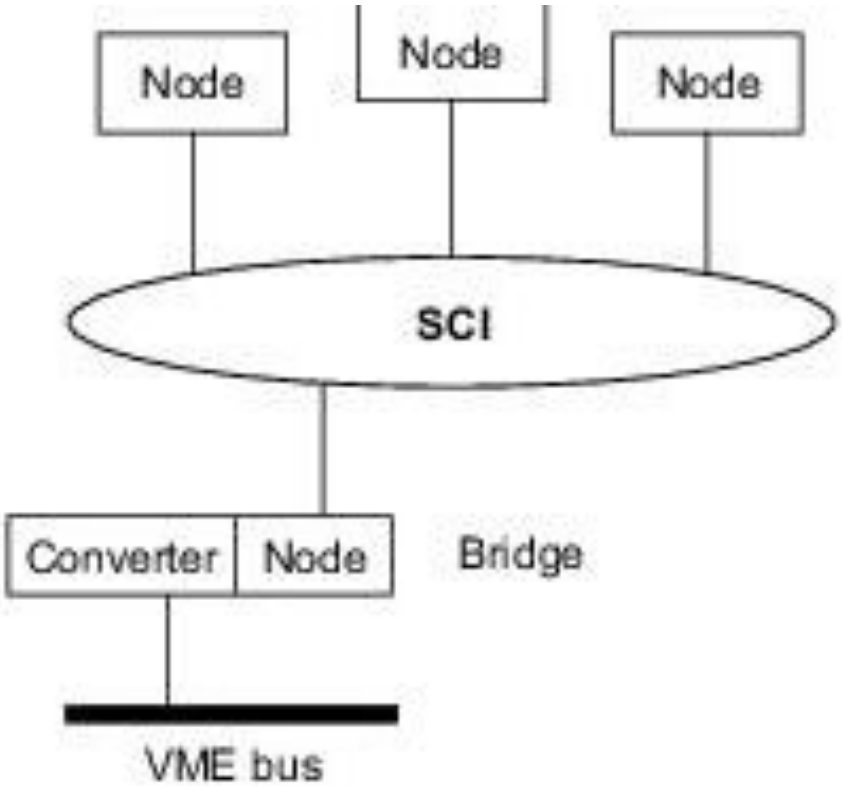


- Reduction in the number of cycles wasted due to read misses
- Magnitude of the benefits varied across the three programs due to different write-hit ratios
- Low hit ratios
- Data set size is large
- Parallelism decreases spatial locality in the application

Scalable Coherence Interface

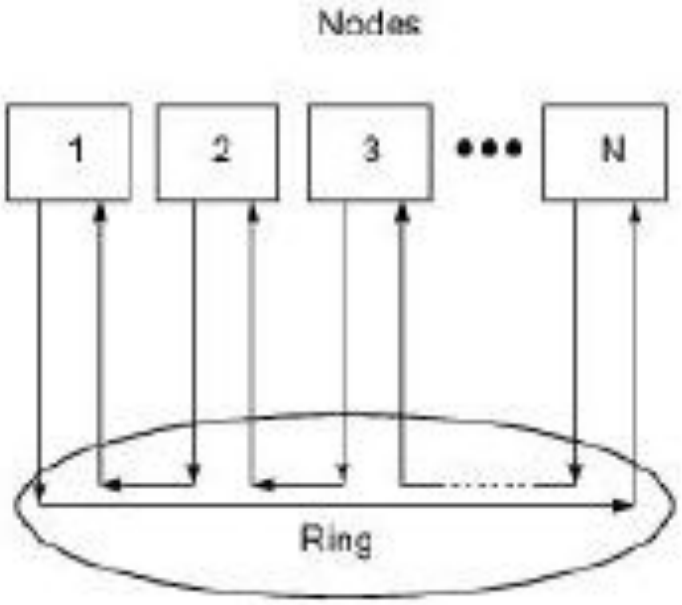
- extend from conventional bused backplanes to a fully duplex, point-to-point interface specification
- Interface standard for very high performance multiprocessors
- SCI defines the interface between nodes and the external interconnect
- I6-bit links with a bandwidth of up to 1 Gbytes per link

Scalable Coherence Interface

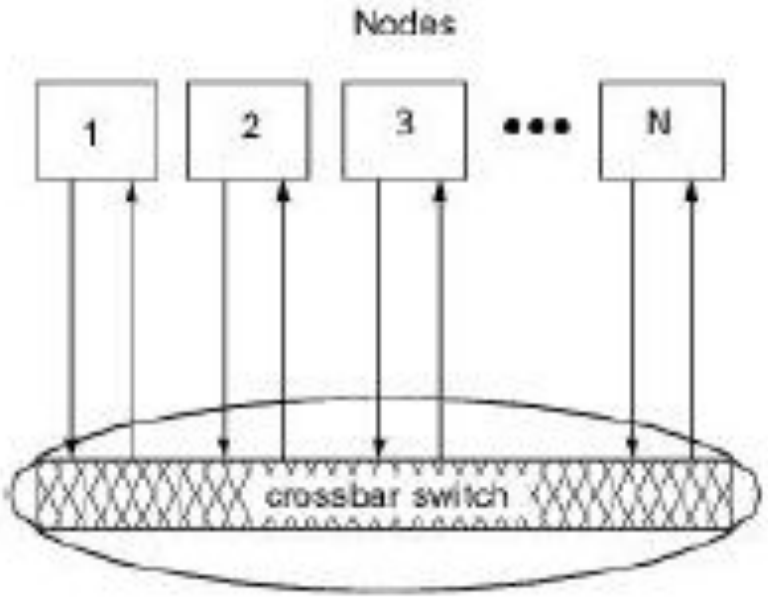


Scalable Coherence Interface

- SCI interconnect models



(b) A ring for point-to-point transactions



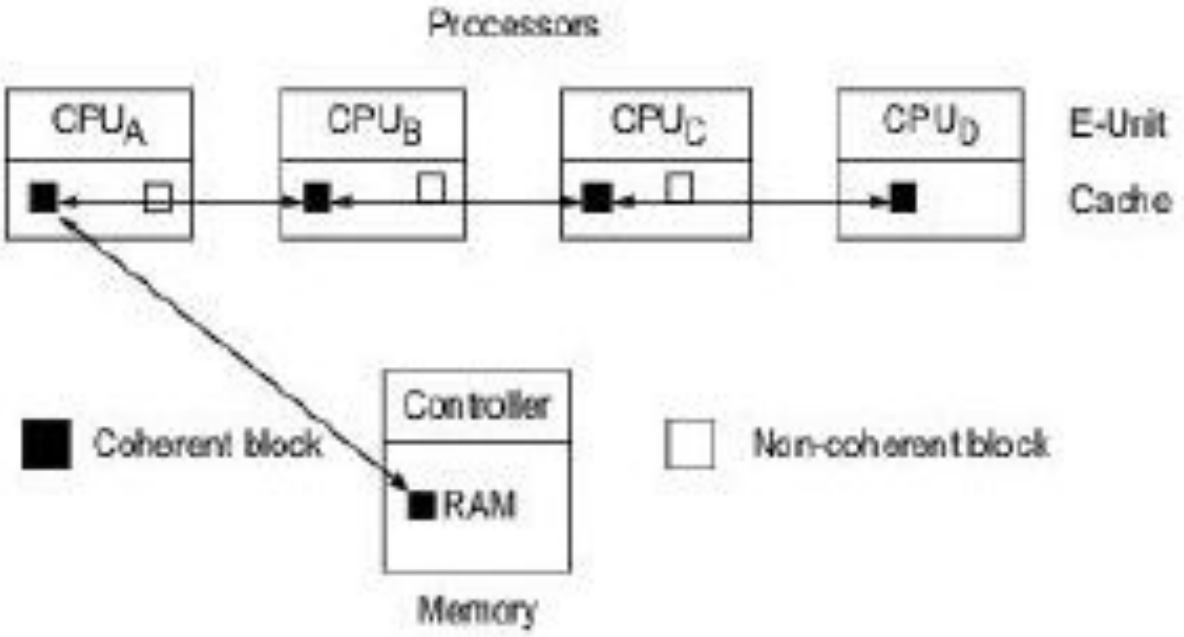
(b) A crossbar multiprocessor

SCI coherence protocol

- Sharing-list structure
  - Sharing lists are used in SCI to build chained directories
  - length is unbounded
  - dynamically created, pruned, and destroyed.

SCI coherence protocol

- Sharing-list structure



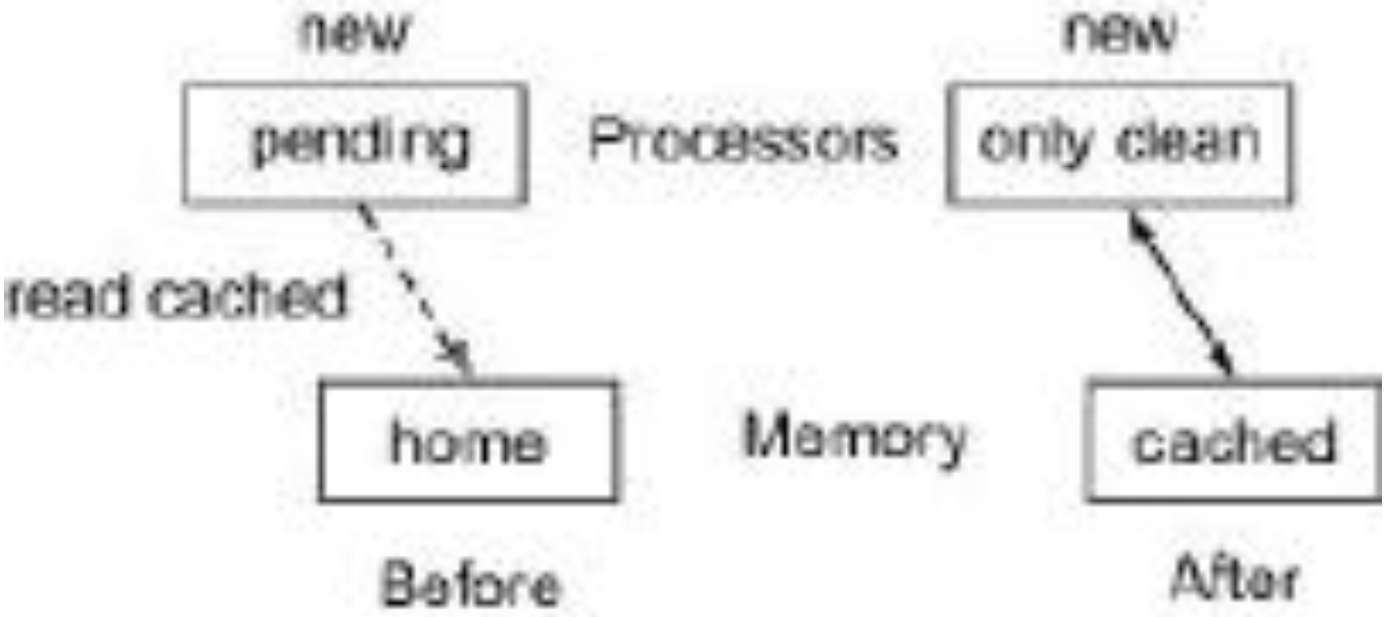
SCI cache coherence protocol with distributed directories

SCI coherence protocol

- Sharing-list creation
- the shared memory is either in a home (uncached)or a cached (sharing-list) state
- The head processor is always responsible for list management

SCI coherence protocol

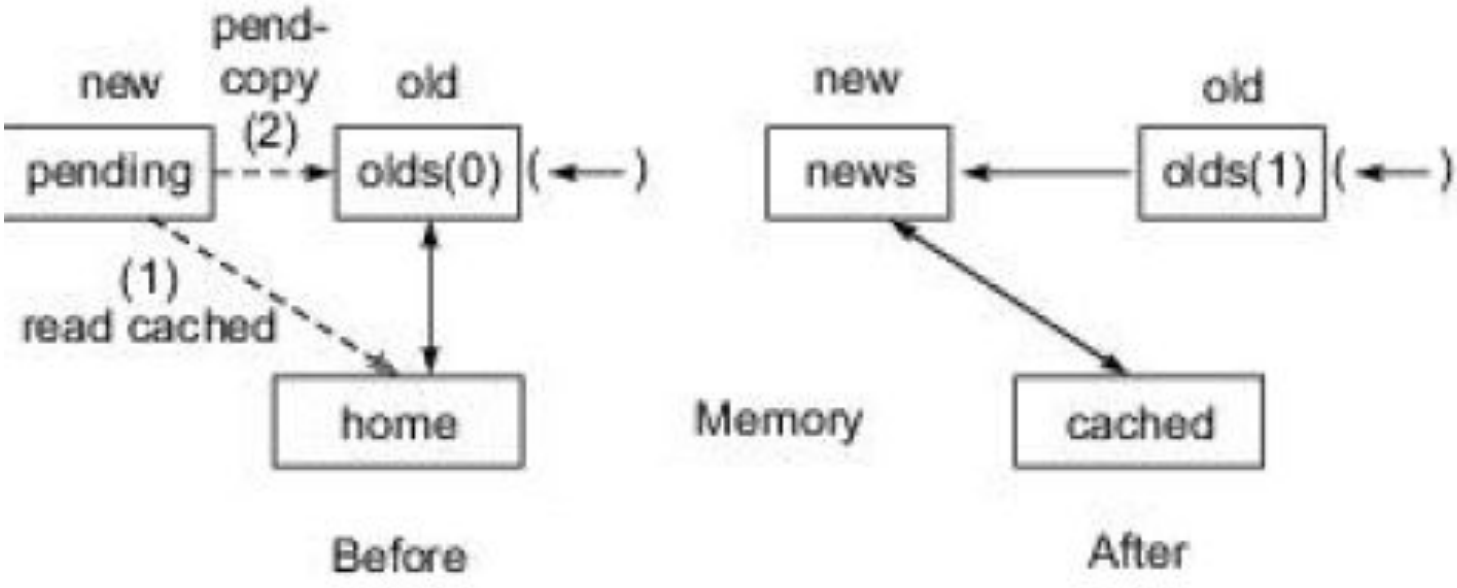
- Sharing-list creation



(a) Creation of sharing list

SCI coherence protocol

Addition of new nodes



(b) Addition of new nodes

Relaxed Memory Consistency

- 2 memory models :
    - **Processor Consistency** : writes issued by each individual processor are always in program order
- Two conditions related to other processors

(1) Before a *read* is allowed to perform with respect to any other processor, all previous read accesses must be performed

(2) Before a *write* is allowed to perform with respect to any other processor, all previous read or write accesses must be performed

- **Release Consistency :** synchronization accesses in the program be identified and classified as either acquires (e.g. locks) or release(e.g. unlocks)

Example 1: Processor Consistent

P_1	W(x)1	W(x)3		
P_2			R(x)1	R(x)3
P_3	W(y)1	W(y)2		
P_4			R(y)1	R(y)2

Example 2: Not Processor Consistent

P_1	W(x)1	W(x)3		
P_2			R(x)3	R(x)1
P_3	W(y)1	W(y)2		
P_4			R(y)2	R(y)1

- Three conditions ensure release consistency:

(1) Before an ordinary read or write access is allowed to perform with respect to any other processor, all previous acquire accesses must be performed

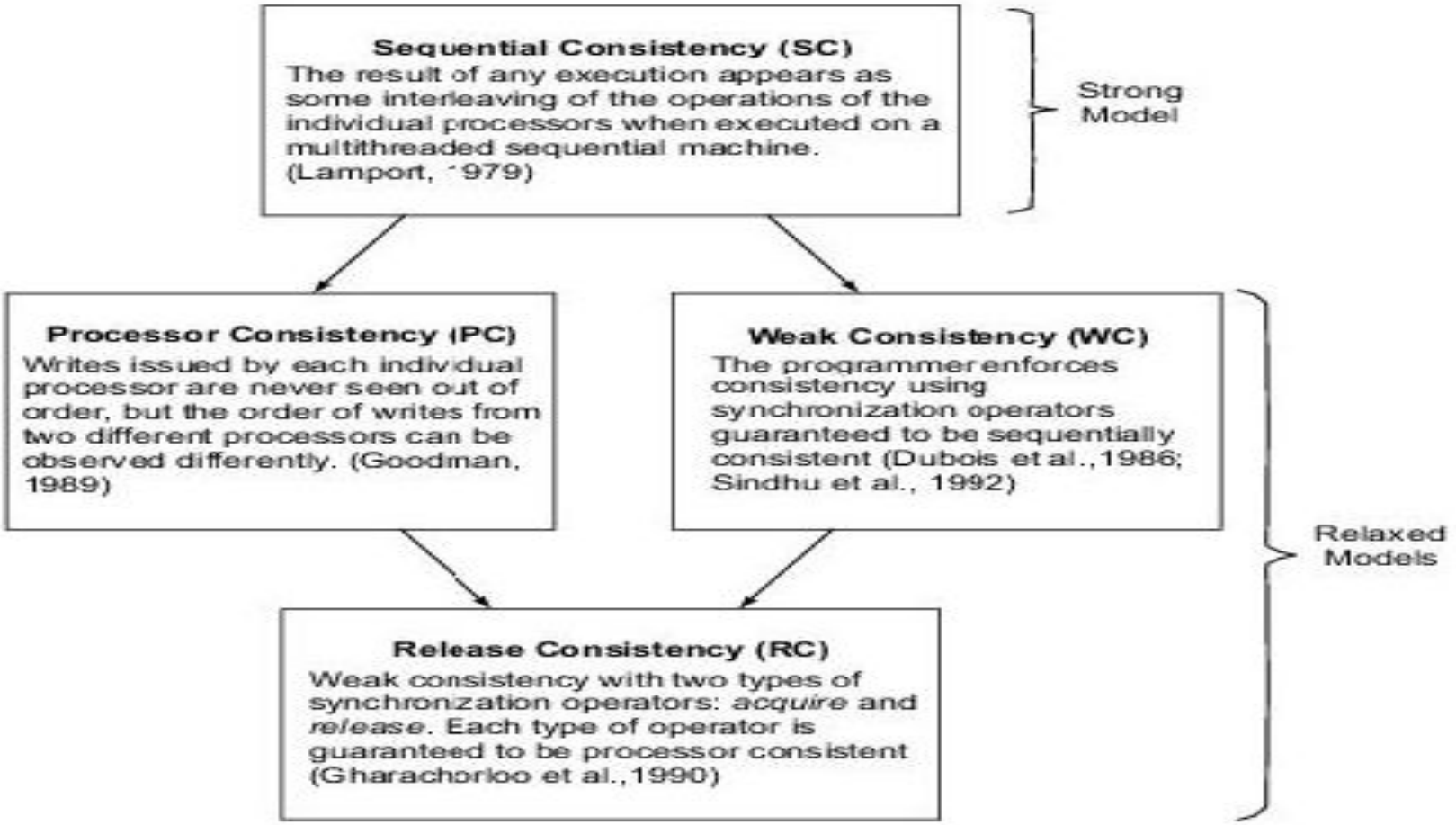
(2) Before a release access is allowed to perform with respect to any other processor all previous ordinary read and store accesses must be performed

(3) Special accesses are processor-consistent with one another. The ordering restrictions imposed by weak consistency arc not present in release consistency. instead, release consistency requires processor consistency and not sequential consistency

- Consistency can be satisfied by

(i) Stalling the processor on an acquire access until it completes

(ii) Delaying the completion of release access until all previous memory accesses complete



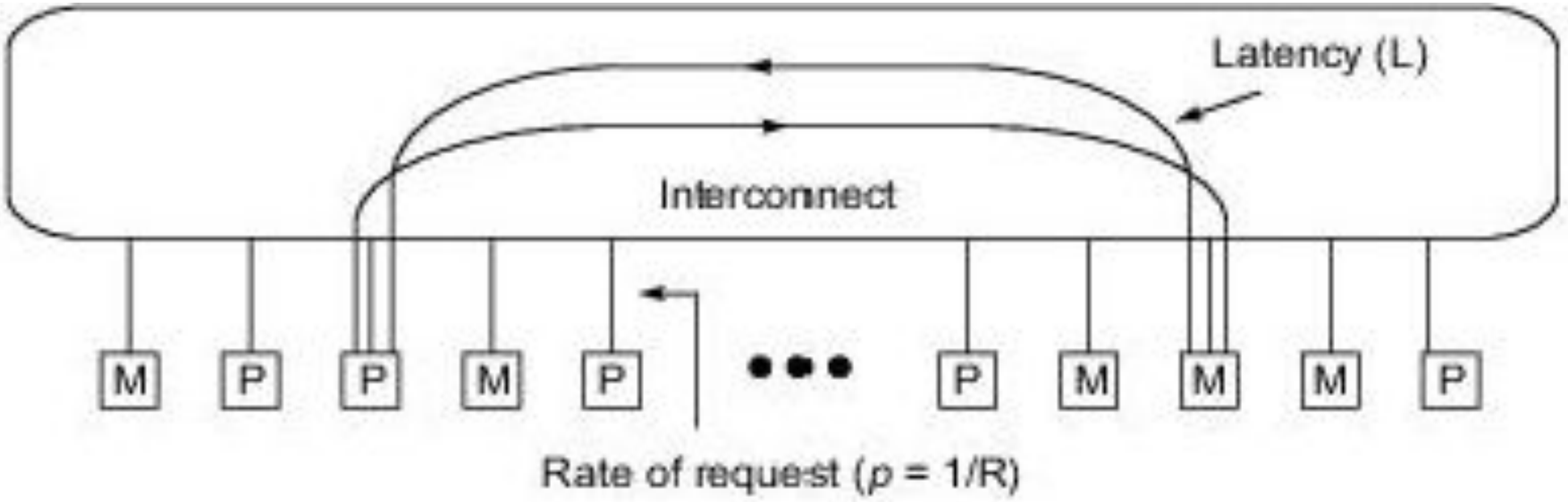
MODULE VI

- Multithreaded and data flow architectures
  - Latency hiding techniques
  - **Principles of multithreading**
    - Multithreading Issues and Solutions
    - Multiple context Processors
  - Fine-grain Multicomputer
    - Fine-grain Parallelism
  - Dataflow and hybrid architecture

Principles Of Multithreading

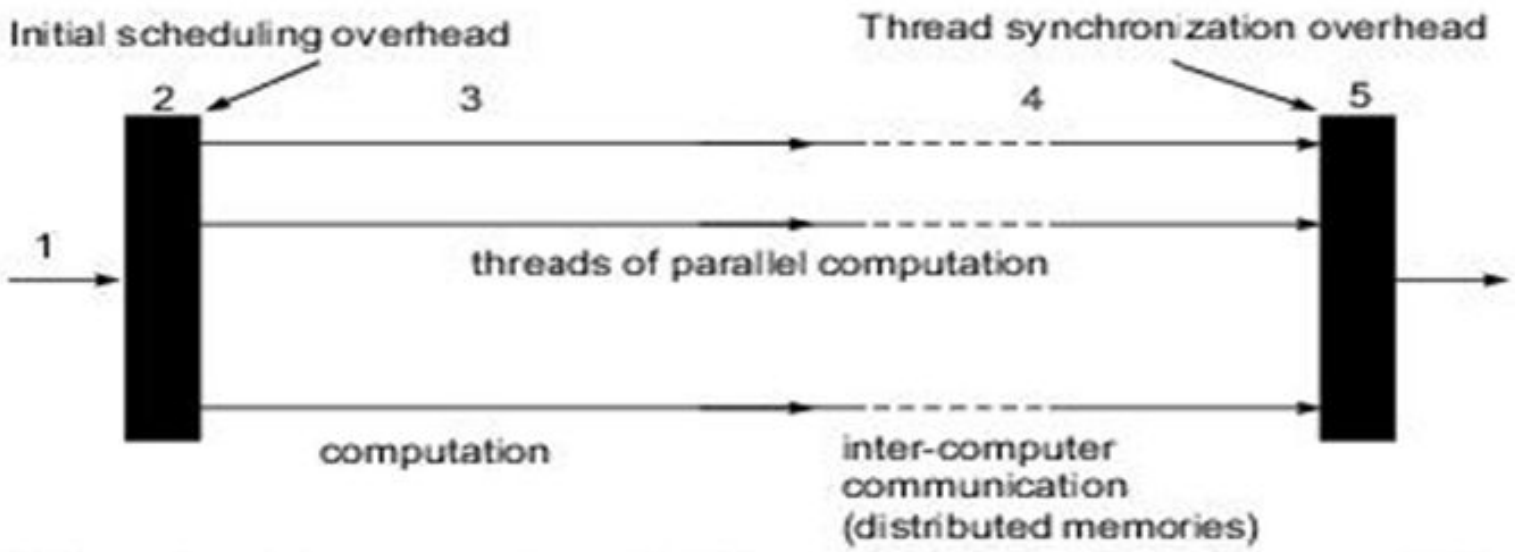
- **Multithreaded Issues and Solutions**
  - Demands that the processor be designed to handle multiple contexts simultaneously on a context-switching basis
  - **Architecture Environment :** Multithreaded MPP system is modeled by a network of processor (P) and memory (M) nodes as depicted in Figure

Principles Of Multithreading



- Four machine parameters to analyze the performance of this network
  - The *latency (L)*: communication latency on a remote memory access
  - The *number of thread* (*N*) : number of threads that can be interleaved in each processor
  - The *Context-switching overhead (C)* : lost in performing contest switching in a processor
- The *Interval between switches (R)* : cycles between switches triggered by remote reference

- Distributed memories form a global address space
  - **Fig** : Multithreaded architecture and its computation model for a massively parallel processing system



(b) Multithreaded computation model. (Courtesy of Gordon Bell *Commun. ACM*, August 1992)

- **Multithreaded Computations**

The computation starts with a sequential thread

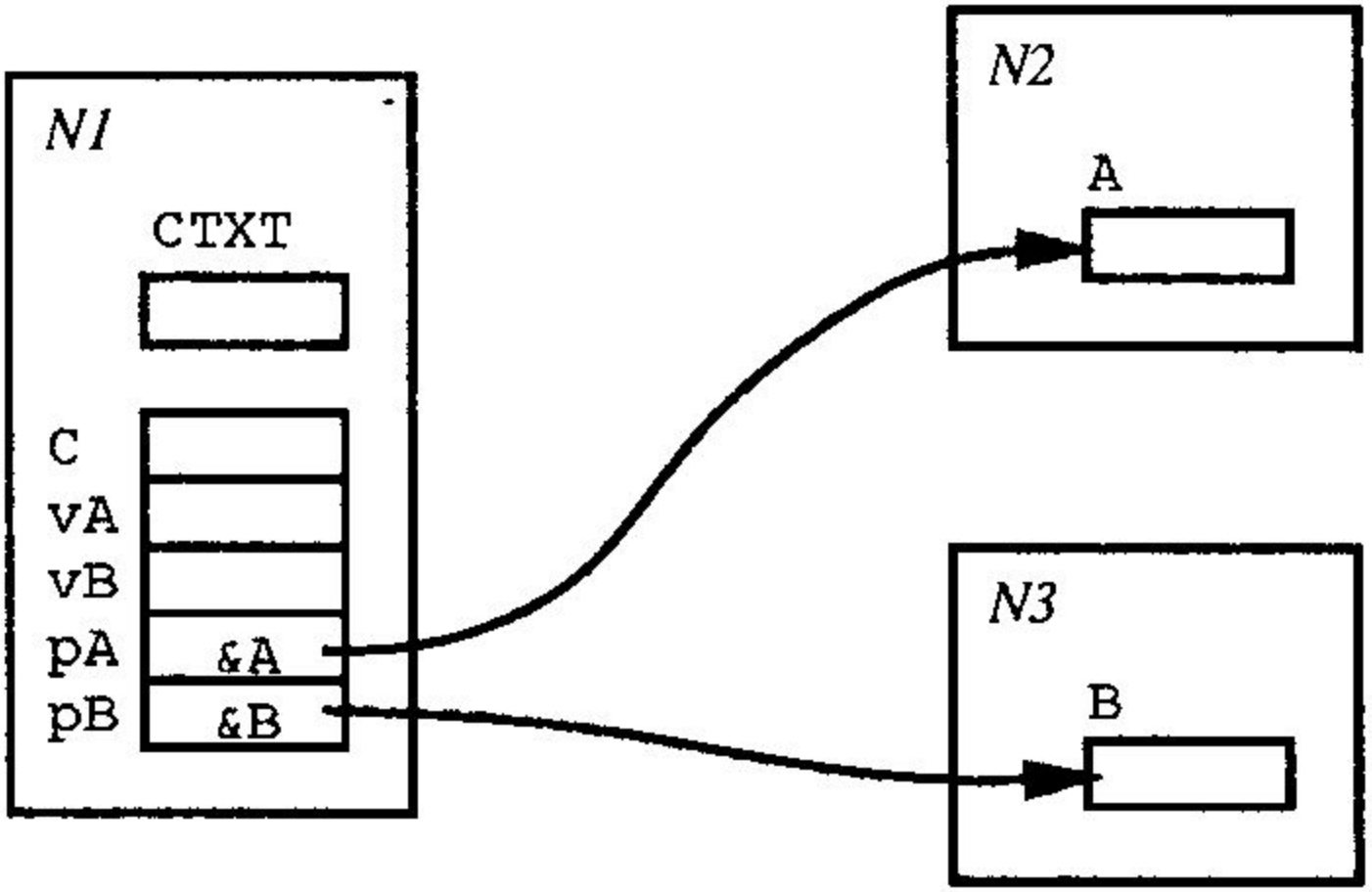
- **Multithreaded Computations**

The computation starts with a sequential thread

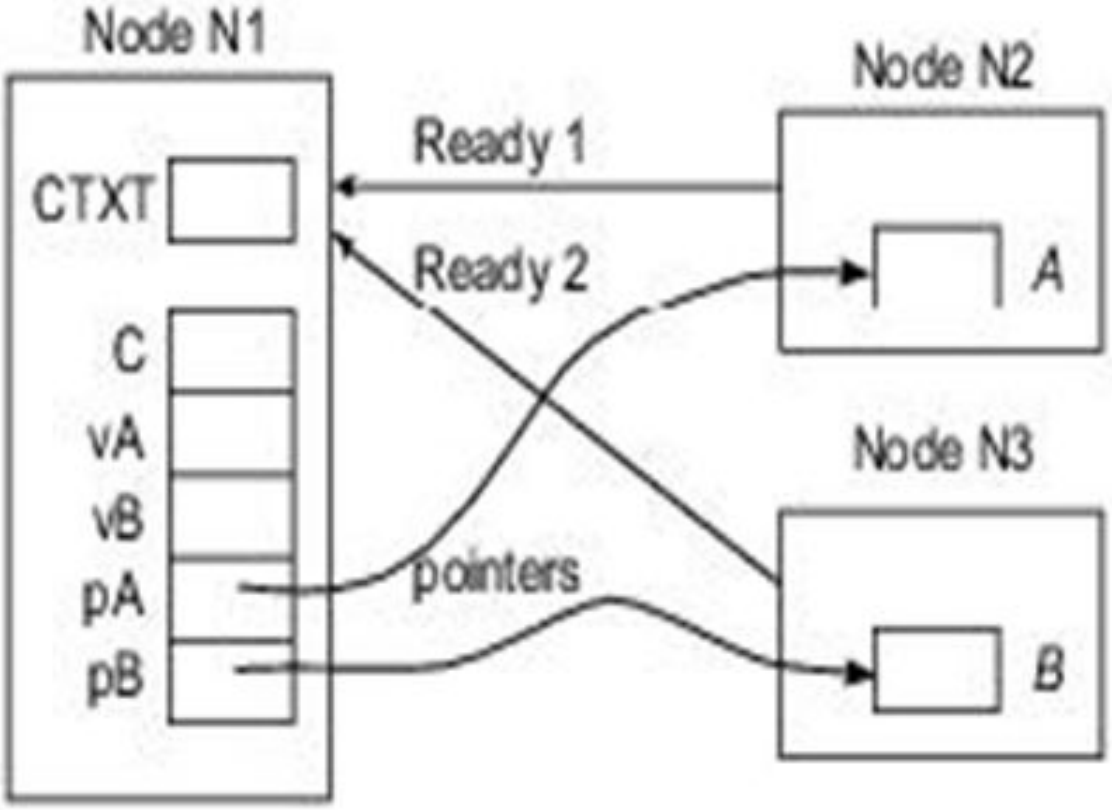
Multithreading issues

- Fundamental latency problems:
  - remote load / Synchronizing load

- **Remote loads problems**



vA = rload pA
vB = rload pB
C = vA - VB



On Node N1, compute: $C = A - B$

A and B computed concurrently

Thread on N1 must be notified when A, B are ready

(b) The synchronizing loads problem

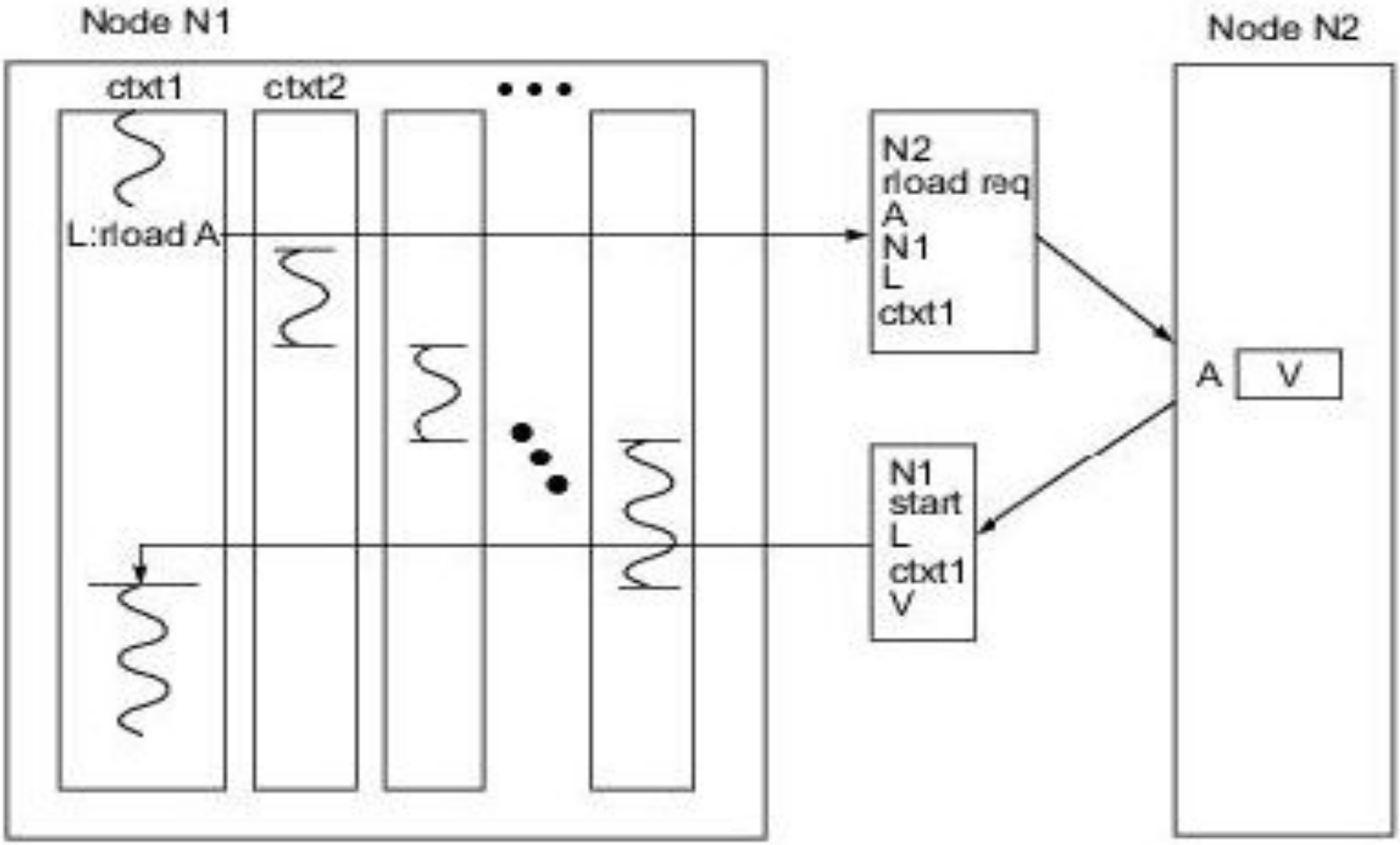
- Remote load situation is illustrated in Fig (a)
- Let pA and pB be the pointers to A and B, respectively
- The two rloads can be issued from the same thread or from two different threads
- The context of the computation on N1 is represented by the variable CTXT
- Variable names like vA, vB, and C are interpreted relative to CTXT

- Fig (b) : idling due to synchronizing loads
- A and B are computed by concurrent processes, and not sure exactly when they will be ready for node N1 to read
- The ready signals (Ready1 and Ready2) may reach node N1 asynchronously
- This is a typical situation in the producer consumer problem
- Busy-waiting may result

- Key issue in remote loads : how to avoid idling in node N1 during the load operations
- The latency caused by synchronizing loads also depends on scheduling and the time it takes to compute A and B
- Synchronization latency is often unpredictable, while the remote-load latencies are often predictable

Multithreading solutions

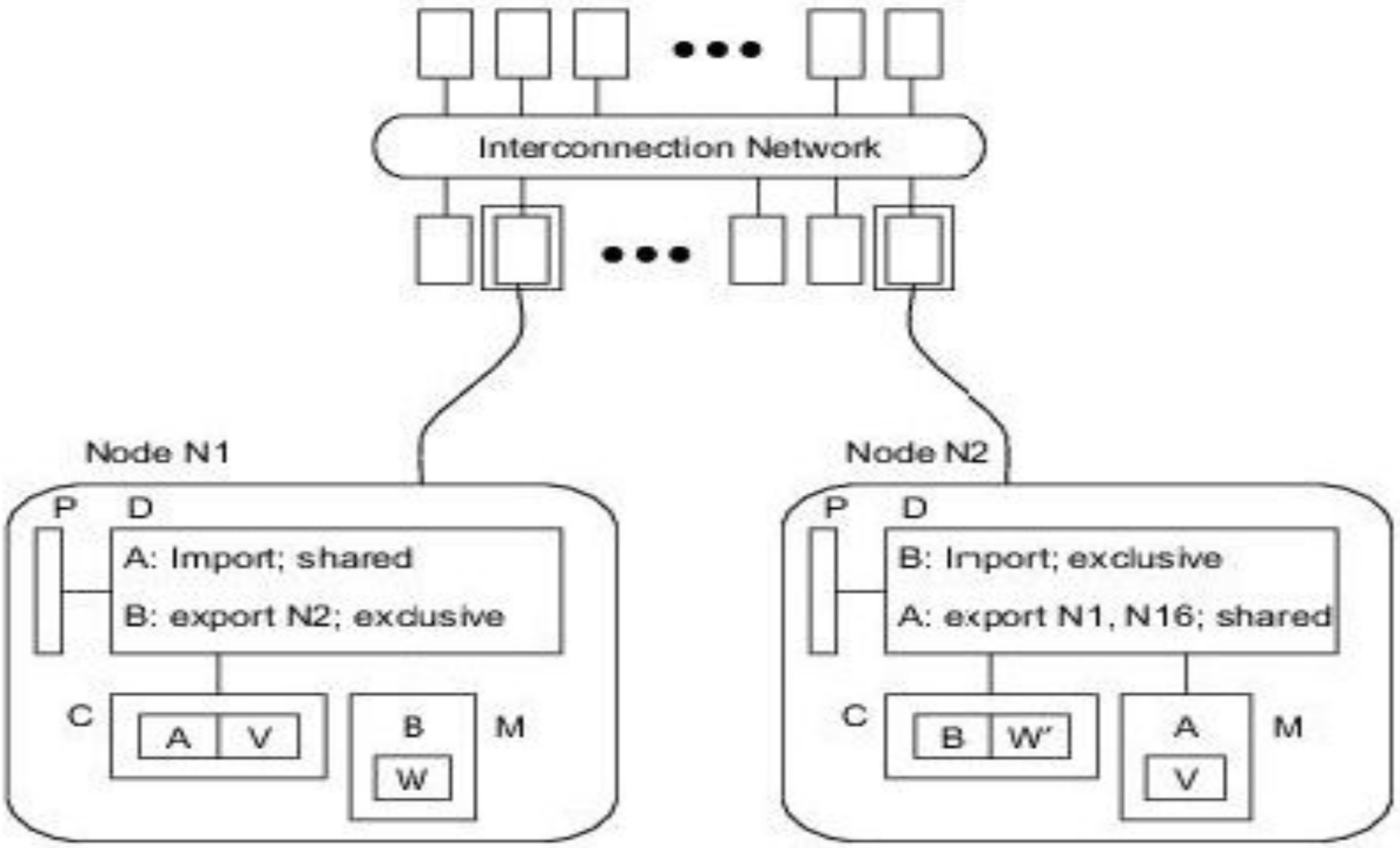
- multiplex among many threads
  - Cost of thread switching should be much smaller than that of the latency of the remote load, or else the processor might as well wait for the remote load's response
  - Make sure that messages carry continuations



(a) Multithreading solution

- As the internode latency increases, more threads are needed to hide it effectively. Another concern is to make sure that messages carry continuations Suppose, after issuing a remote load from thread T1 we switch to thread E, which also issues a remote load.
- The responses may not return in the same order. This may be caused by requests traveling different distances, through varying degrees of congestion, to destination nodes whose loads differ greatly, etc.
- One way to cope with the problem is to associate each remote load and response with an identifier for the appropriate thread, so that it can be reenabled on the arrival of a response. These thread identifiers are referred to as continuations on messages,

- Distributed Cacheing



P = Processor; D = Directory; C = Cache; M = Memory

(b) Distributed cacheing

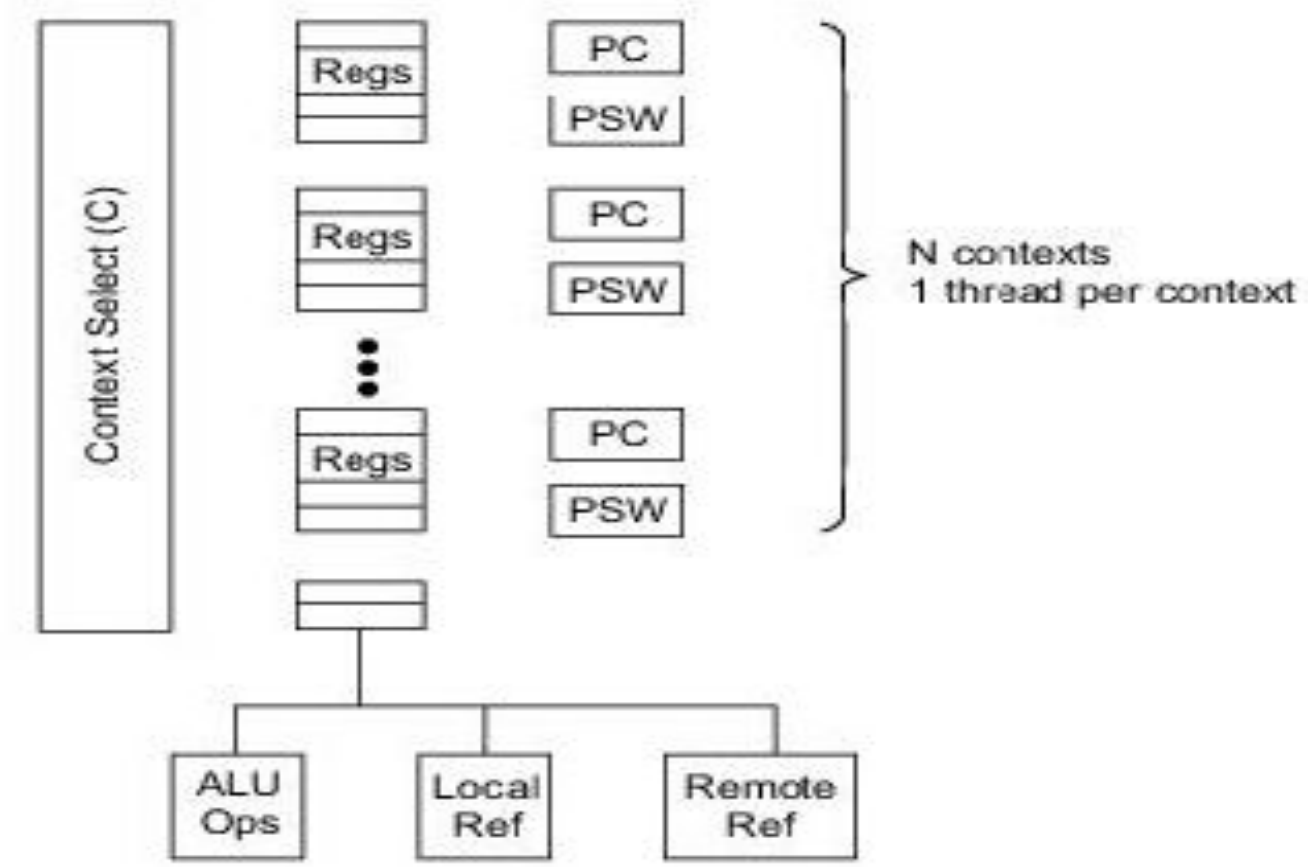
- Every memory location has an owner node
- For example, N1 owns B and N2 owns A
- The directories are used to contain import-export lists and state whether the data is shared or exclusive

Multiple-Context Processors

- Multithreaded systems are constructed with multiple-context(or multithreaded) processors
- processor efficiency issue as a function of memory latency (L), the number of contexts (N), and context-switching overhead (C)

$$Efficiency = \frac{busy}{busy + switching + idle}$$

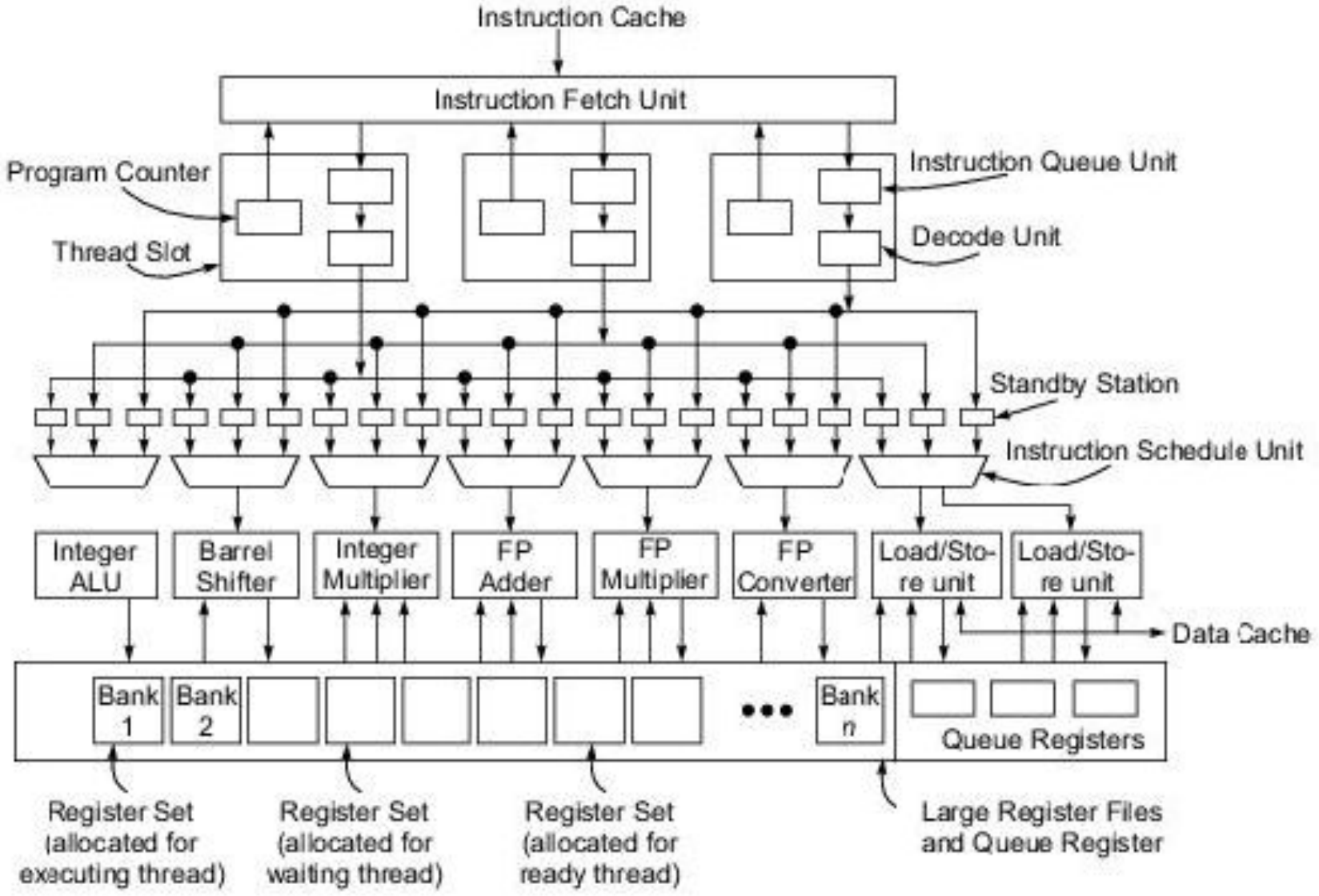
- a context cycles through the following states: ready, running , leaving and blocking



(a) Multithreaded model. (Courtesy of Rafael Saavedra, 1992)

Example 9.2 A multithreaded processor with three thread slots (Hiroaki Hirata et al.,1992)

- Processor is provided with several instruction queue unit and decode unit pairs called thread slots
- Each thread slot, associated with a program counter, makes up a logical processor, while an instruction fetch unit and all functional units are physically shared among logical processors(**Fig (b)**)



(b) A three-thread processor example (Courtesy of H. Hirata et al, *Proc 19th Int. Symp. Comput. Archit.*, Australia, May 1992)

- Instruction queue unit has a buffer
- Buffer size needs to be at least $B = N * C$ words
- N : no. Of thread slots
- C : no. Of cycles required to access

- Context-Switching Policies:

(1) *Switching on cache miss* - context is preempted when it causes a cache miss. R is taken to be the average interval between misses (in cycles), and L the time required to -satisfy the miss

(2) *Switching on every load* - policy allows switching on every load. L independent of whether it will cause a miss or not. R represents the average interval between loads

(3) *Switching on every instruction* - This policy allows switching on every instruction, independent of whether it is a load or not

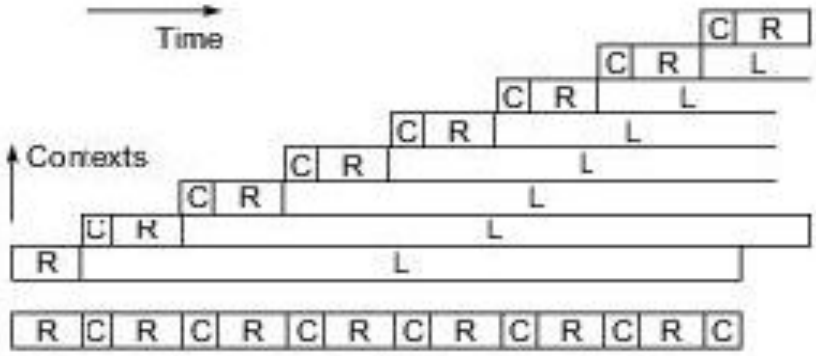
- *Switching on block of instruction* - Blocks of instructions from different threads are interleaved. This will improve the cache-hit ratio due to locality.

Processor Efficiencies : Alternating renewal process having a cycle of R + L

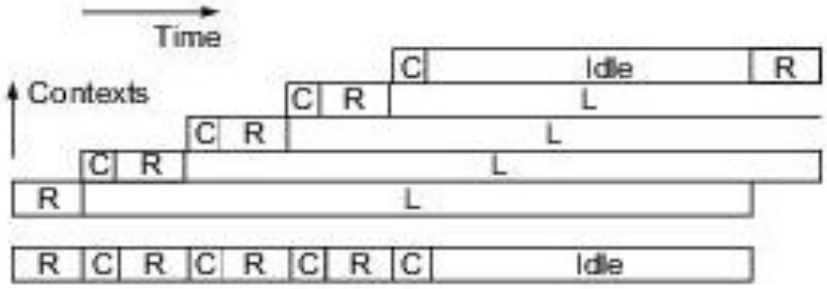
- Efficiency of a single - threaded machine is given by

$$E1 = \frac{R}{R+L} = \frac{1}{1+L/R}$$

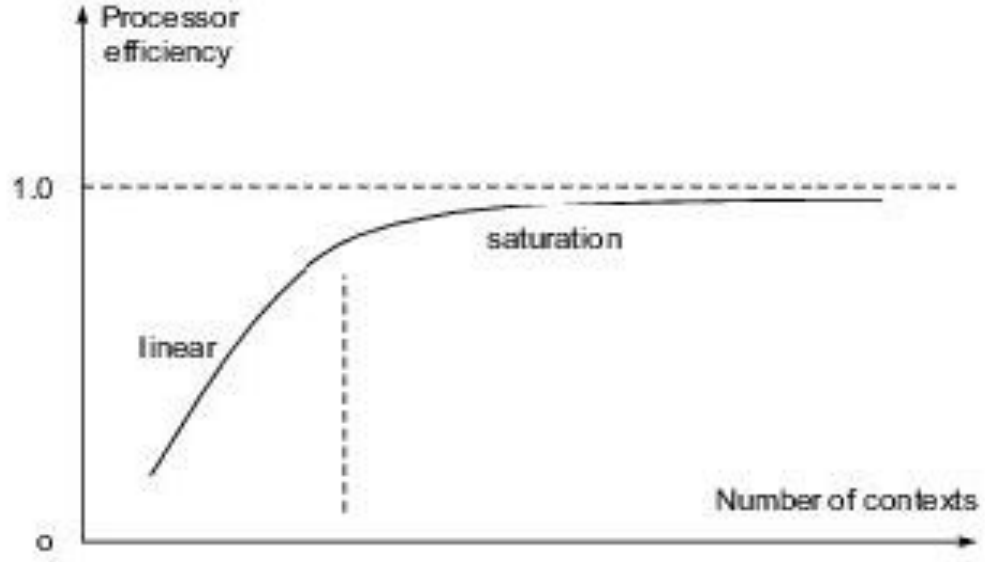
Fig : Context switching and processor efficiency as a function of the number of context



(a) Snapshots of context switching in the saturation region



(a) Snapshots of context switching in the linear region



(c) Efficiency curve

(1) Saturation region – In this saturated region, the processor operates with maximum utilization. The cycle of the renewal process in this -case is R + C, and the efficiency is simply

$$E_{sat.} = \frac{R}{R + C} = \frac{1}{1 + C/R}$$

(2) Linear region — When the number of contexts is below the saturation point, there may be no ready contests alter a context switch, so the processor will experience idle cycles

$$E_{in.} = \frac{NR}{R + C + L}$$

MODULE VI

- Multithreaded and data flow architectures
  - Latency hiding techniques
  - Principles of multithreading
    - Multithreading Issues and Solutions
    - Multiple context Processors
  - **Fine-grain Multicomputer**
    - Fine-grain Parallelism
  - Dataflow and hybrid architecture

Fine-grained parallelism

- In parallel computing granularity of a task is a measure of the amount of work (or computation) which is performed by that task.
- Granularity is usually measured in terms of the number of instructions executed in a particular task
- **Fine-grained parallelism**
- **Coarse-grained parallelism**
- **Medium-grained parallelism**

Fine-grained parallelism

- In fine-grained parallelism, a program is broken down to a large number of small tasks.
- fine-grained parallelism facilitates load balancing

coarse-grained parallelism

- In coarse-grained parallelism, a program is split into large tasks.
- a large amount of computation takes place in processors. - result in load imbalance

Medium-grained parallelism

- Medium-grained parallelism is a compromise between fine-grained and coarse-grained parallelism

Fine-grain Multicomputers

Cray Y-MP were used to perform coarse-grain computations Message-passing multicomputers are used to execute medium-grain programs

Large grain size implies lower concurrency or lower DOP

Fine grain –higher DOP

Fine-Grain Parallelism

- Grain sizes, communication latencies, and concurrency in four classes of parallel computers
- **Latency Analysis** : Fine-Grain, Medium-Grain and Course-Grain Machine characteristics of Some examples systems(Table)

Characteristics	*Machine*			
	Cray Y-MP	*Connection Machine CM-2*	*Intel iPSC/1*	*MIT J-Machine*
Communication latency, T_c	40 ns via shared memory	600 μs per 32-bit *send* operation	5 ms	2 μs
Synchronization overhead, T_s	20 μs	125 ns per bit-slice operation in lock step	500 μs	1 μs
Grain size, T_g	20 s	4 μs per 32-bit result per PE instruction	10 ms	5 μs
Concurrency (DOP)	2–16	8K–64K	8–128	1K–64K
Remark	Coarse-grain supercomputer	Fine-grain data parallelism	Medium-grain multicomputer	Fine-grain multicomputer

MODULE VI

- Multithreaded and data flow architectures
  - Latency hiding techniques
  - Principles of multithreading
    - Multithreading Issues and Solutions
    - Multiple context Processors
  - Fine-grain Multicomputer
    - Fine-grain Parallelism
  - **Dataflow and hybrid architecture**

Dataflow and hybrid architecture

- **The Evolution of Dataflow Computers :**
  - Memory latency and synchronization overhead as two fundamental issues in multiprocessing
  - Dataflow architectures represent a radical alternative to von Neumann architectures because they use dataflow graphs as their machine languages
  - Specify only a partial order for the execution of instructions and thus provide opportunities for parallel and pipelined execution at the level of individual instructions

Execution of instruction, a=(b+1)*(b-c) in data flow computers.

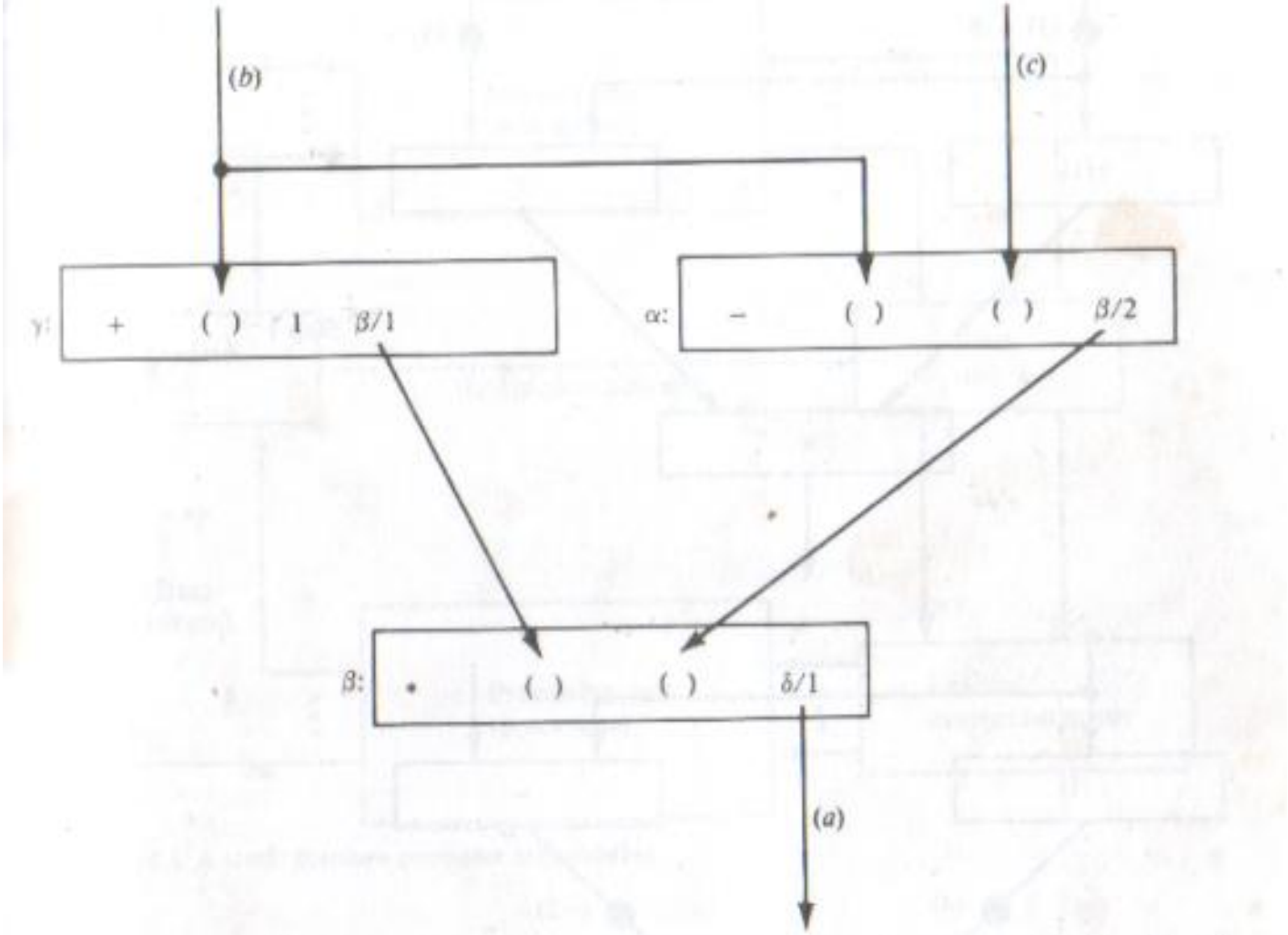
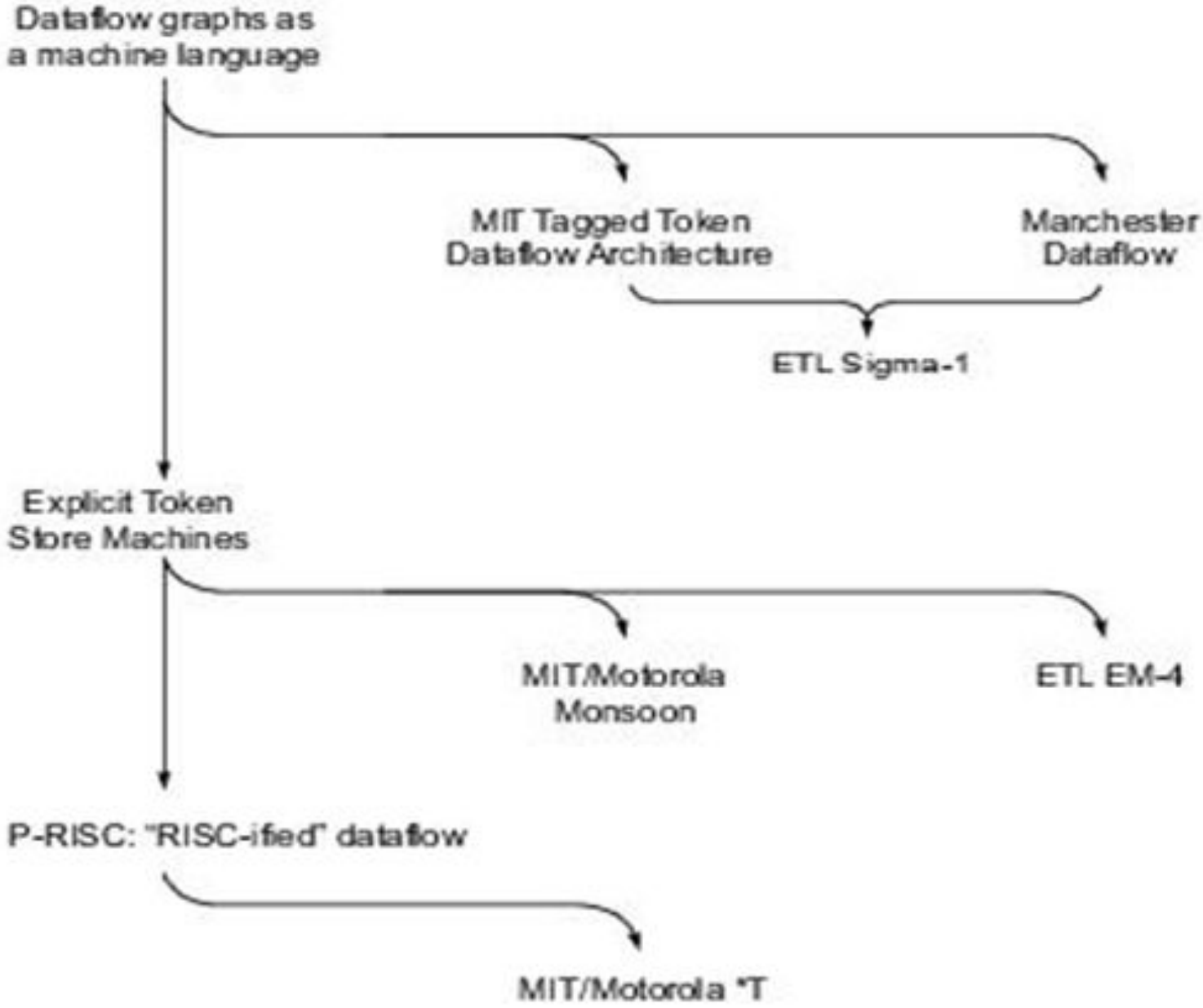
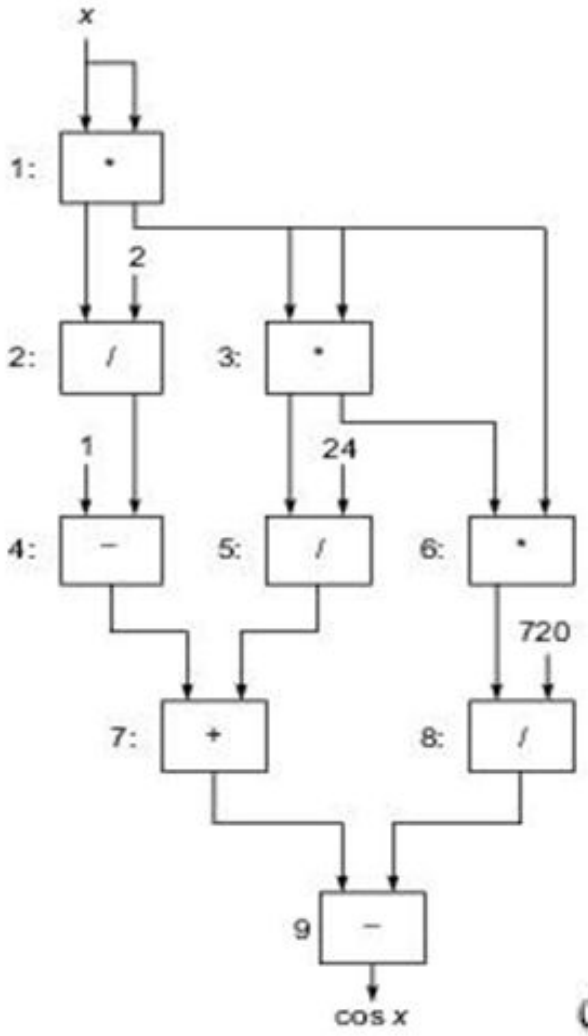


Figure 10.2 Instruction execution in a dataflow computer for the computation of $a = (b + 1) * (b - c)$ by direct data forwarding.


Dataflow graphs as
a machine language
MIT Tagged Token
Dataflow Architecture
Manchester
Dataflow
ETL Sigma-1
Explicit Token
Store Machines
MIT/Motorola
Monsoon
ETL EM-4
P-RISC: "RISC-ified" dataflow
MIT/Motorola *T

Dataflow Graphs

- Used as a machine language in dataflow computers



$$\cos x \simeq 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720}$$

(a) Dataflow graph for computing cos x (Courtesy of Arvind)

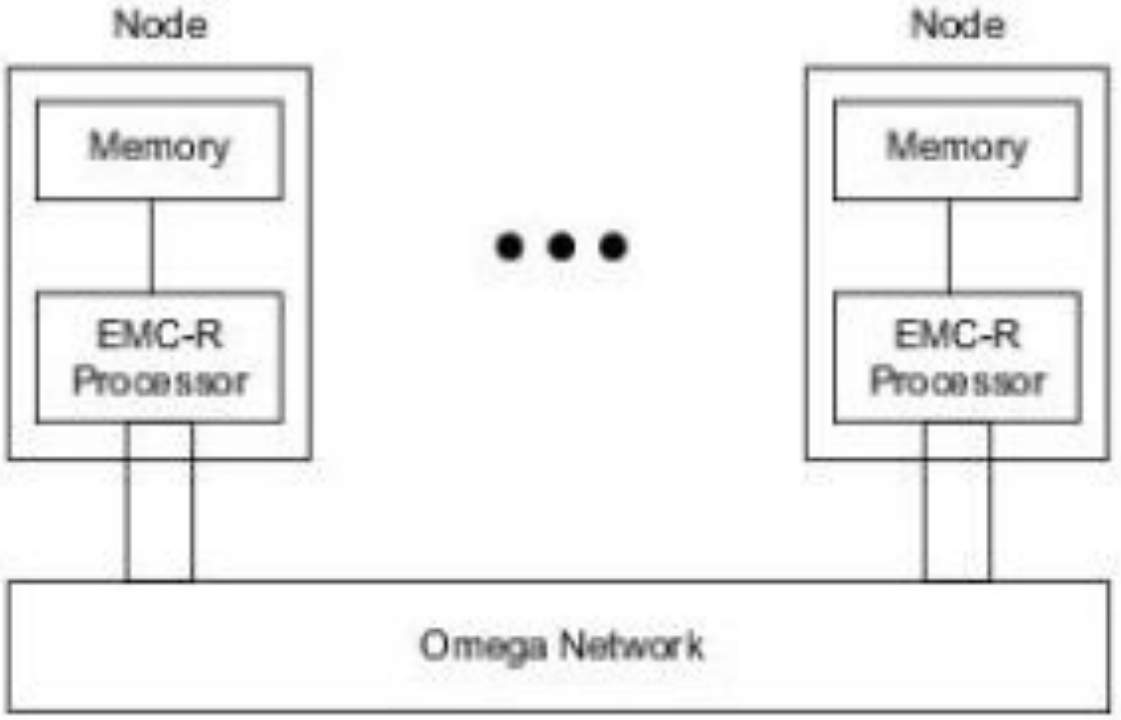
Static versus Dynamic Dataflow:

- Disallow more than one token to reside on any one arch
- A node is enabled as soon as tokens are present on all input arcs and there is no token on any of its output arcs.
- *Acknowledge Signals*

- In a dynamic architecture, each data token is tagged with a contest descriptor, called a *tagged token*
- Pure Dataflow Machines
- Explicit Token Store Machines
- Hybrid and Unified iirchitectures

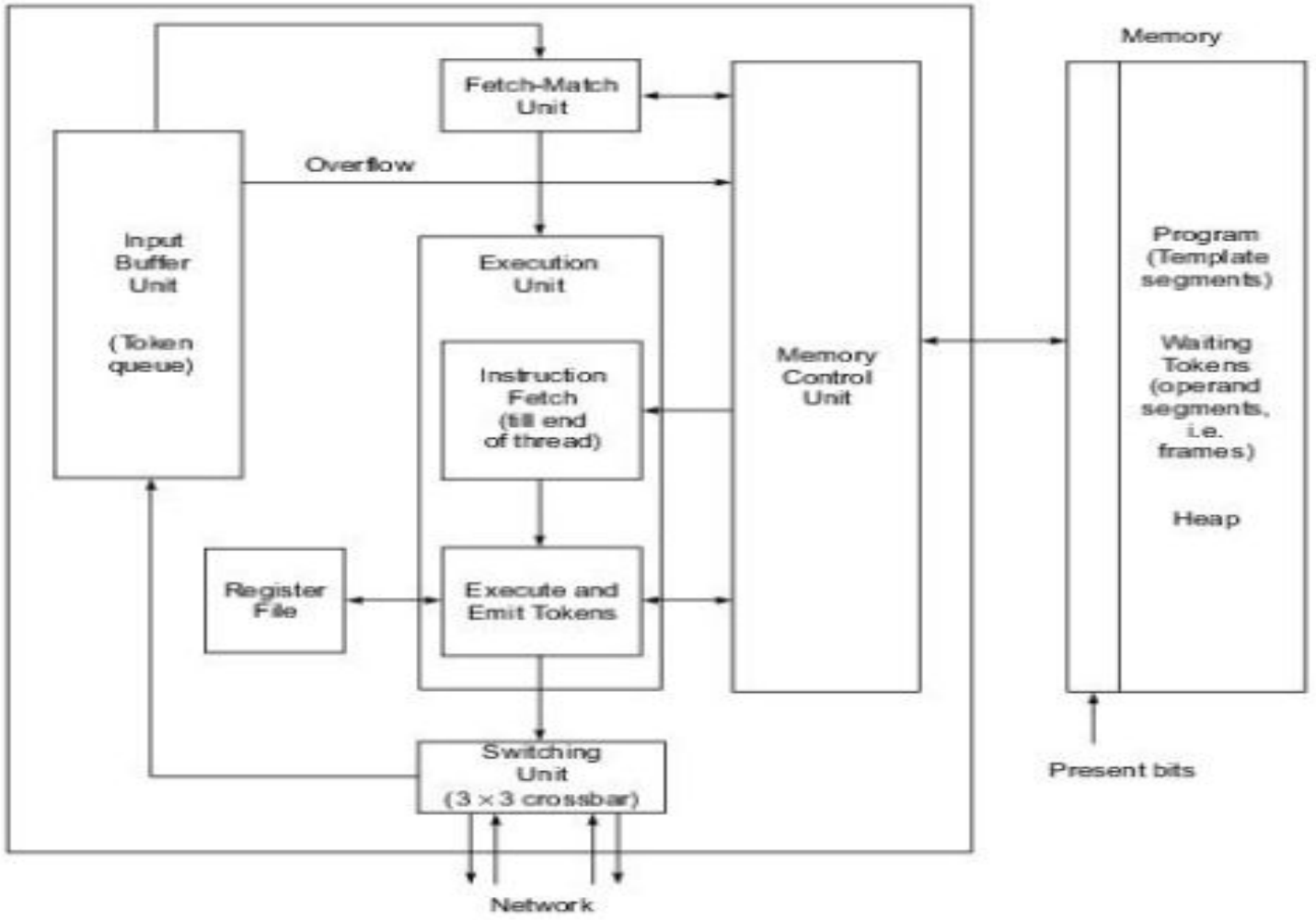
ETL/EM-4 in Japan

- EM-4 had the overall system organization



(a) Global organization

- The Node Architecture : internal design of the processor chip and of the node memory

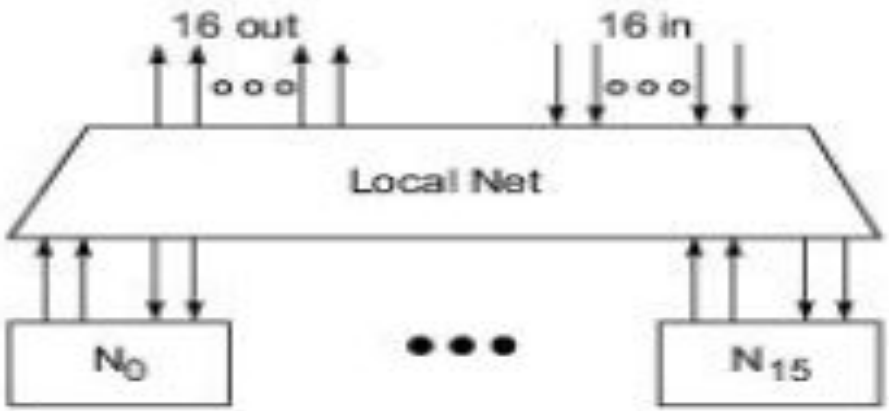


(b) The EMC-R processor design

- 3 * 3 cross bar *switch unit*
- *Memory control unit*
- *Input buffer*
- *Fetch-match unit*
- *Execution unit*
- *Full/empty* bits

The MIT/Motorola *T Prototype

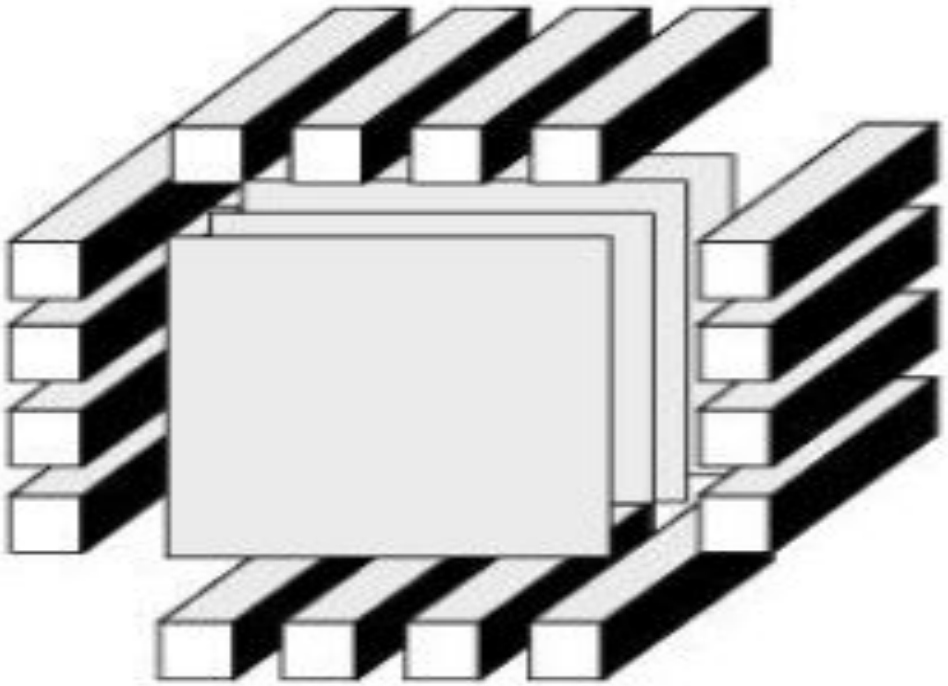
- *T project was a direct descendant of a series of MIT dynamic dataflow architectures unifying with the von Neumann architectures
- **The Prototype Architecture :**
  - *T prototype was a single-address-space system
  - A “brick" of 16 nodes was packaged in a 9-in cube



(a) A brick of 16 nodes with 3.2-Gflops and 3200-MIPS peak performance, packaged in a 9-in cube

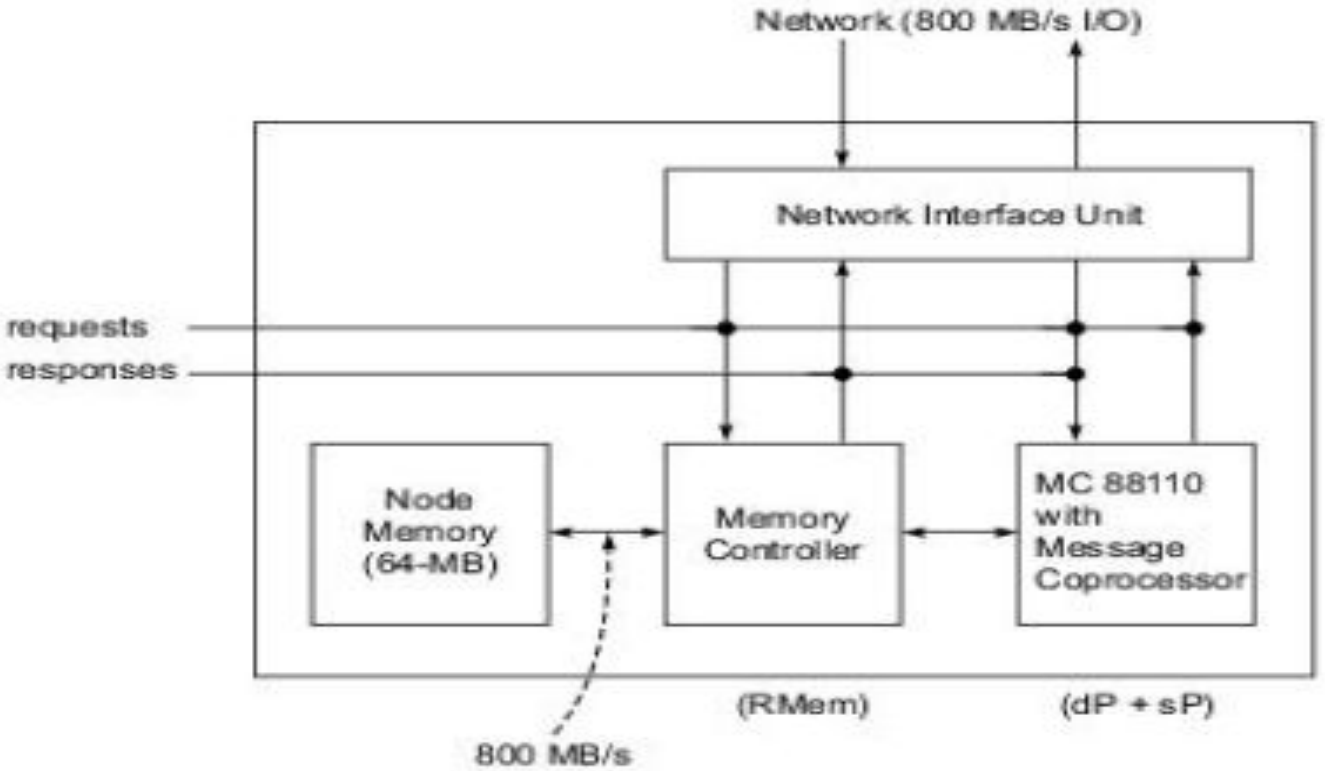
- Local network was built with 8*8 crossbar switching chips
- Memory was distributed to the nodes
- One gigabyte of RAM was used per brick
- 200-mbytes/s links, the I/0 bandwidth was 6.4 gbytes/s per brick

- 256-node machine could be built with 16 bricks



(b) A 256-node machine consisting of 16 bricks interconnected by 4 boards of 16 × 16 switches and packaged in a 1.5-m cube

- **The *T Node Design :** Motorola superscalar RISC microprocessor (MC88110) was modified as a data processor (dP)
- Synchronization coprocessor (sP)
- Memory Controller
- Network interface unit



(c) Interior node architecture with data processor (MC 88110) and synchronization coprocessor (sP)

- I-structure semantics was also implemented in *T
- Full/empty bits were used on producer consumer variables
- *T treated messages as virtual continuations
- **Multithreading : A Perspective**
  - The Dash, KSR-1, and Alewife leveraged existing processor technology
  - Less aggressive pursuit of parallelism and depend heavily on compilers to obtain locality
  - In von Neumann multithreading approaches, the HEP/Tera replicated the conventional instruction stream
- In the dataflow approaches, the system-level view has stayed constant from the Tagged-Token Dataflow architecture to the *T